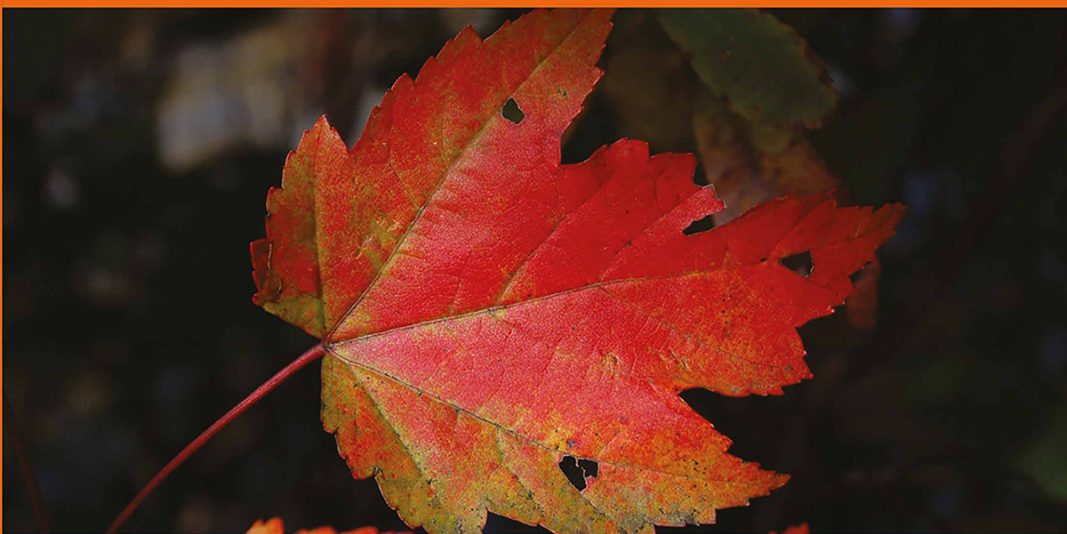


Projetos de Ciência de Dados com Python

Abordagem de estudo de caso para a criação de projetos de ciência de dados bem-sucedidos usando Python, pandas e scikit-learn



Packt
novatec

Stephen Klosterman

VISIT...

LANZAROTE
Caliente.COM

Projetos de Ciência de Dados com Python

Abordagem de estudo de caso para a criação de projetos de ciência de dados bem-sucedidos usando Python, pandas e scikit-learn



Packt
novatec

Stephen Klosterman

Projetos de Ciência de Dados com Python

**Abordagem de estudo de caso para a
criação de projetos de ciência de
dados bem-sucedidos usando Python,
pandas e scikit-learn**

Stephen Klosterman

Packt
Novatec

Copyright © Packt Publishing 2019. First published in the english language under the title 'Data Science Projects with Python – (9781838551025)'

Copyright © Packt Publishing 2019. Publicação original em inglês intitulada 'Data Science Projects with Python – (9781838551025)'

Esta tradução é publicada e vendida com a permissão da Packt Publishing.

© Novatec Editora Ltda. [2019].

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Tradução: Aldir Coelho Corrêa da Silva

Revisão gramatical: Tássia Carvalho

Editoração eletrônica: Carolina Kuwabata

ISBN do ebook: 978-65-86057-11-9

ISBN do impresso: 978-65-86057-10-2

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: novatec@novatec.com.br

Site: www.novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Sumário

Prefácio

capítulo 1 ■ Exploração e limpeza de dados

Introdução

Python e o sistema de gerenciamento de pacotes Anaconda

A indexação e o operador slice

Exercício 1: Examinando o Anaconda e familiarizando-se com o Python

Diferentes tipos de problemas da ciência de dados

Carregando os dados do estudo de caso com o Jupyter e o pandas

Exercício 2: Carregando os dados do estudo de caso em um Jupyter Notebook

Familiarizando-se com os dados e executando sua limpeza

O problema da empresa

Etapas da exploração de dados

Exercício 3: Verificando a integridade básica dos dados

Máscaras booleanas

Exercício 4: Continuando a verificação da integridade dos dados

Exercício 5: Explorando e limpando os dados

Exploração e garantia da qualidade dos dados

Exercício 6: Explorando o limite de crédito e as características demográficas

Aprofundamento nas características categóricas

Exercício 7: Implementando a OHE para uma característica categórica

Explorando as características de histórico financeiro do dataset

Atividade 1: Explorando as características financeiras restantes do dataset

Resumo

capítulo 2 ■ Introdução ao Scikit-Learn e avaliação do modelo

Introdução

Examinando a variável de resposta e concluindo a exploração inicial

Introdução ao scikit-learn

Gerando dados sintéticos

Dados para uma regressão linear

Exercício 8: Regressão linear com o scikit-Learn

Métricas de desempenho de modelos para a classificação binária

Dividindo os dados: conjuntos de treinamento e de teste

Acurácia da classificação

Taxa de verdadeiros positivos, taxa de falsos positivos e matriz de confusão

Exercício 9: Calculando as taxas de verdadeiros e falsos positivos e negativos e a matriz de confusão em Python

Descobrimos probabilidades previstas: como a regressão logística faz previsões?

Exercício 10: Obtendo probabilidades previstas a partir de um modelo de regressão logística

Curva receiver operating characteristic (ROC)

Precisão

Atividade 2: Executando a regressão logística com uma nova característica e criando uma curva precision-recall

Resumo

capítulo 3 ■ Detalhes da regressão logística e exploração de características

Introdução

Examinando os relacionamentos entre as características e a resposta

Correlação de Pearson

Teste F

Exercício 11: Teste F e seleção de características univariada

Pontos mais importantes do teste F: equivalência com o teste t para duas classes e cuidados

Hipóteses e próximas etapas

Exercício 12: Visualizando o relacionamento entre as características e a resposta

Seleção de características univariada: o que ela pode ou não fazer

Entendendo a regressão logística com sintaxe de funções Python e a função sigmóide

Exercício 13: Plotando a função sigmóide

Escopo das funções

Por que a regressão logística é considerada um modelo linear?

Exercício 14: Examinando a conveniência das características para a regressão logística

Dos coeficientes da regressão logística às previsões com o uso da função sigmóide

Exercício 15: Limite de decisão linear da regressão logística

Atividade 3: Ajustando um modelo de regressão logística e usando os coeficientes diretamente

Resumo

capítulo 4 ■ O trade-off entre viés e variância

Introdução

Estimando os coeficientes e as intercepções da regressão logística

Gradiente descendente para a descoberta de valores de parâmetros ótimos

Exercício 16: Usando o gradiente descendente para reduzir a função custo

Suposições da regressão logística

Motivação para a regularização: O trade-off entre viés e variância

Exercício 17: Gerando e modelando dados de classificação sintéticos

Regularização lasso (L1) e ridge (L2)

Validação cruzada: Selecionando o parâmetro de regularização e outros hiperparâmetros

Exercício 18: Reduzindo o overfitting no problema de classificação de dados sintéticos

Opções da regressão logística no scikit-learn

Escalonamento de dados, pipelines e características de interação no scikit-learn

Atividade 4: Validação cruzada e engenharia de características com os dados do estudo de caso

Resumo

capítulo 5 ■ Árvores de decisão e florestas aleatórias

Objetivos do aprendizado

Introdução

Árvores de decisão

Terminologia das árvores de decisão e conexões com o Machine Learning

Exercício 19: Uma árvore de decisão no scikit-learn

Treinando árvores de decisão: Impureza dos nós

Características usadas para as primeiras divisões: Conexões com a seleção de características univariada e as interações

Treinando árvores de decisão: Um algoritmo ganancioso

Treinando árvores de decisão: Diferentes critérios de parada

- Usando árvores de decisão: Vantagens e probabilidades previstas
- Abordagem mais conveniente da validação cruzada
- Exercício 20: Encontrando hiperparâmetros ótimos para uma árvore de decisão
- Florestas aleatórias: Combinações de árvores de decisão
- Floresta aleatória: Previsões e interpretabilidade
- Exercício 21: Ajustando uma floresta aleatória
- Gráfico quadriculado (checkerboard)
- Atividade 5: Busca em grade na validação cruzada com floresta aleatória

Resumo

capítulo 6 ■ Imputação de dados faltantes, análise financeira e distribuição para o cliente

Objetivos do aprendizado

Introdução

Revisão dos resultados dos modelos

Lidando com dados faltantes: Estratégias de imputação

- Preparando amostras com dados faltantes
- Exercício 22: Limpando o dataset
- Exercício 23: Imputação de PAY_1 pela moda e aleatória
- Um modelo preditivo para PAY_1
- Exercício 24: Construindo um modelo de classificação multiclasse para a imputação
- Usando o modelo de imputação e comparando-o com outros métodos
- Confirmando o desempenho do modelo com o conjunto de teste desconhecido
- Análise financeira
- Conversa financeira com o cliente
- Exercício 25: Caracterizando custos e economias
- Atividade 6: Derivando insights financeiros

Considerações finais sobre a distribuição do modelo preditivo para o cliente

Resumo

Apêndice

Capítulo 1: Exploração e limpeza de dados

- Atividade 1: Explorando as características financeiras restantes do dataset

Capítulo 2: Introdução ao Scikit-Learn e avaliação do modelo

- Atividade 2: Executando a regressão logística com uma nova característica e criando uma curva precision-recall

Capítulo 3: Detalhes da regressão logística e exploração de características

- Atividade 3: Ajustando um modelo de regressão logística e usando os coeficientes diretamente

Capítulo 4: O trade-off entre viés e variância

- Atividade 4: Validação cruzada e engenharia de características com os dados do estudo de caso

Capítulo 5: Árvores de decisão e florestas aleatórias

- Atividade 5: Busca em grade na validação cruzada com floresta aleatória

Capítulo 6: Imputação de dados faltantes, análise financeira e distribuição para o cliente

- Atividade 6: Derivando insights financeiros

Prefácio

Esta seção apresentará de maneira resumida o autor, a abordagem usada pelo livro, as habilidades técnicas iniciais necessárias e o hardware e os programas requeridos para a execução de todas as atividades e exercícios fornecidos.

Sobre o livro

Projetos de ciência de dados com Python foi projetado para oferecer orientação prática sobre ferramentas padrão da indústria para análise de dados e machine learning em Python com a ajuda de dados reais. O livro o ajudará a entender como usar pandas e o Matplotlib para examinar criticamente um dataset com sínteses estatísticas e gráficos e extrair insights significativos. Você continuará adquirindo conhecimento ao aprender a preparar dados e a fornecê-los para algoritmos de machine learning, como o de regressão logística regularizada e o de floresta aleatória (random forest), usando o pacote scikit-learn. Também aprenderá como ajustar algoritmos para fornecer as melhores previsões sobre dados novos não conhecidos. À medida que percorrer os capítulos mais avançados, conhecerá o funcionamento e a saída desses algoritmos e entenderá melhor não só os recursos preditivos dos modelos, mas também a maneira como eles fazem previsões.

Sobre o autor

Stephen Klosterman é um cientista de dados da área de machine learning na CVS Health, o qual ajuda a acomodar os problemas dentro do contexto da ciência de dados e fornece soluções de machine learning que os stakeholders empresariais entendem e valorizam. Sua formação inclui um Ph.D. em biologia na Universidade de Harvard, onde foi professor assistente do curso de ciência de dados.

Objetivos

- Instalar os pacotes necessários para a definição de um ambiente de codificação de ciência de dados
- Carregar dados em um Jupyter Notebook executando Python
- Usar o Matplotlib para criar visualizações de dados
- Criar um modelo usando o scikit-learn
- Usar o lasso e a regressão ridge para reduzir o overfitting (sobreajuste)
- Criar e ajustar um modelo de floresta aleatória e comparar o desempenho com o da regressão logística
- Criar visualizações usando a saída do Jupyter Notebook

Público-alvo

Se você é analista de dados, cientista de dados ou analista empresarial e deseja aprender a usar Python e técnicas de machine learning para analisar dados e prever resultados, este livro o ajudará. É aconselhável ter conhecimento básico em programação de computadores e análise de dados. Familiaridade com conceitos matemáticos como álgebra e estatística básica também será útil.

Abordagem

Projetos de ciência de dados com Python usa a abordagem de estudo de caso para simular as condições de trabalho que você encontrará ao aplicar conceitos de ciência de dados e machine learning. Serão apresentados um problema e um dataset, e você percorrerá as etapas de definir a questão a ser solucionada, escolher os métodos de análise a serem usados e implementar tudo isso em Python para chegar a um resultado.

Requisitos de hardware

Para o aluno ter uma experiência ideal, recomendamos a configuração de hardware a seguir:

- Processador: Intel Core i5 ou equivalente

- Memória: 4 GB de RAM
- Armazenamento: 35 GB de espaço disponível

Requisitos de software

Você também precisará dos programas a seguir já instalados:

- Sistema operacional: Windows 7 SP1 de 64 bits, Windows 8.1 de 64-bits ou Windows 10 de 64 bits, Ubuntu Linux, ou a versão mais recente do OS X
- Navegador: Versão mais recente do Google Chrome/Mozilla Firefox
- Notepad++/Sublime Text como IDE (esse requisito é opcional, já que você pode fazer tudo usando o Jupyter Notebook em seu navegador)
- O Python a partir da versão 3.4 (a mais recente é a 3.7) tem que estar instalado (pode ser baixado de <https://python.org>)
- Bibliotecas Python conforme necessário (Jupyter, NumPy, Pandas, Matplotlib e assim por diante)

Instalação e configuração

Antes de começar a ler o livro, certifique-se de instalar o ambiente Anaconda já que usaremos essa distribuição do Python.

Instalando o Anaconda

Instale o Anaconda seguindo as instruções deste link: <https://www.anaconda.com/distribution/>

Recursos adicionais

O code bundle deste livro está hospedado no GitHub em <https://github.com/TrainingByPackt/Data-Science-Projects-with-Python>.

Há outros code bundles de nosso rico catálogo de livros e vídeos disponíveis em <https://github.com/PacktPublishing/>. Acesse!

Convenções

Palavras de códigos contidas no texto, nomes de tabelas de bancos de dados, nomes de pastas, nomes de arquivos, extensões de arquivos, nomes de caminhos, URLs fictícias, entradas do usuário e handles do Twitter serão exibidos da seguinte forma: “Digitando conda list na linha de comando, você verá todos os pacotes que estão instalados em seu ambiente”.

Um bloco de código será exibido assim:

```
import numpy as np #cálculo numérico
import pandas as pd #preparação de dados (data wrangling)
import matplotlib.pyplot as plt #pacote de plotagem
# A linha a seguir ajuda a renderizar plotagens
%matplotlib inline
import matplotlib as mpl #adiciona o recurso de plotagem
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
import graphviz #para visualizar árvores de decisão
```

Termos novos e palavras importantes serão exibidos em negrito. Palavras vistas na tela, por exemplo, em menus ou caixas de diálogo, aparecerão assim no texto: “Crie um novo notebook Python 3 a partir do menu **New** como mostrado”.

Exploração e limpeza de dados

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Executar operações básicas em Python
- Descrever o contexto empresarial dos dados do estudo de caso e sua adequação à tarefa
- Executar operações de limpeza de dados
- Examinar sínteses estatísticas e visualizar os dados do estudo de caso
- Implementar a codificação one-hot (one-hot encoding) em variáveis categóricas

Este capítulo é uma introdução às operações básicas em Python e lhe mostrará como executar operações relacionadas a dados tal qual a verificação e a limpeza de dados, a conversão de tipos de dados, a verificação de sínteses estatísticas e muito mais.

Introdução

A maioria das empresas possui uma grande riqueza de dados sobre suas operações e clientes. Gerar relatórios sobre esses dados na forma de diagramas descritivos, gráficos e tabelas é uma boa maneira de entender o estado atual dos negócios. No entanto, para fornecermos orientações quantitativas para estratégias empresariais futuras, é preciso ir um pouco além. É nesse momento que as práticas de machine learning e modelagem preditiva entram em cena. Neste livro, mostraremos como passar de análises descritivas a orientações concretas para operações futuras usando modelos preditivos.

Para atingir esse objetivo, apresentaremos algumas das ferramentas de machine learning mais usadas com o emprego de Python e muitos de

seus pacotes. Você também conhecerá as habilidades práticas necessárias para executar projetos bem-sucedidos: a curiosidade ao examinar dados e a comunicação com o cliente. O tempo gasto examinando um dataset detalhadamente e verificando criticamente se ele atende com precisão à finalidade pretendida não é desperdício. Aqui você aprenderá várias técnicas para avaliar a qualidade dos dados.

Neste capítulo, após nos familiarizarmos com as ferramentas básicas para a exploração de dados, discutiremos alguns cenários de trabalho típicos mostrando como eles podem ser recebidos. Em seguida, começaremos uma exploração completa do dataset de um estudo de caso e o ajudaremos a aprender como descobrir possíveis problemas, para que, quando você estiver pronto para modelar, prossiga com confiança.

Python e o sistema de gerenciamento de pacotes Anaconda

Neste livro, usaremos a linguagem de programação Python, uma linguagem de ponta para o trabalho com ciência de dados, sendo uma das linguagens de programação cujo crescimento está ocorrendo com mais rapidez. Uma razão normalmente citada para a popularidade do Python é que ele é fácil de aprender. Se você tem experiência no uso de Python, ótimo; porém, se tem experiência no uso de outras linguagens, como C, Matlab ou R, não terá dificuldades para usar Python. É preciso conhecer as estruturas gerais da programação de computadores para aproveitar ao máximo este livro. Alguns exemplos dessas estruturas seriam os loops for e as instruções if que guiam o **fluxo de controle** de um programa. Independentemente da linguagem que você tem usado, provavelmente está familiarizado com essas estruturas e também as encontrará em Python.

Um recurso-chave do Python, que é diferente de algumas outras linguagens, é o fato de ele ser indexado a partir de zero; em outras palavras, o primeiro elemento de uma coleção ordenada tem índice igual a 0. O Python também dá suporte à indexação negativa, em que o índice 1 referencia o último elemento de uma coleção ordenada e os índices negativos são contados de trás para a frente a partir do fim. O operador slice, :, pode ser usado para a seleção de múltiplos elementos de uma coleção ordenada dentro de um intervalo, começando do

início ou indo até o fim da coleção.

A indexação e o operador slice

Demonstraremos aqui como a indexação e o operador slice funcionam. Para termos algo para indexar, criaremos uma **lista**, ou seja, uma coleção ordenada **mutável** que pode conter qualquer tipo de dado, inclusive tipos numéricos e strings. A palavra “mutável” significa apenas que os elementos da lista podem ser alterados após serem atribuídos. Para criar os números de nossa lista, que serão inteiros consecutivos, usaremos a função Python interna `range()`. Tecnicamente, a função `range()` cria um **iterador** que converteremos em uma lista usando a função `list()`, mas não se preocupe com esse detalhe. O screenshot a seguir mostra uma lista básica sendo exibida no console:

```
>>> example_list = list(range(1,6))
>>> example_list
[1, 2, 3, 4, 5]
>>> example_list[0]
1
>>> example_list[-1]
5
>>> example_list[-2]
4
>>> example_list[:3]
[1, 2, 3]
>>> example_list[0] = 'a string'
>>> example_list
['a string', 2, 3, 4, 5]
>>> █
```

Figura 1.1 – Criação e indexação de lista.

Algumas coisas que devemos observar na *Figura 1.1*: o ponto final do intervalo está aberto tanto para a indexação de slice quanto para a função `range()`, enquanto o ponto inicial está fechado. Em outras palavras, observe que, quando especificamos o início e o fim de `range()`, o ponto final 6 não foi incluído no resultado, mas o ponto inicial 1 foi. Da mesma forma, ao indexar a lista com o slice `[:3]`, incluímos todos os elementos com índice até 3, porém sem incluir esse último.

Fizemos referência às coleções ordenadas, mas o Python também inclui coleções não ordenadas. Um dos mais importantes chama-se **dicionário**. Um dicionário é uma coleção não ordenada de pares **chave:valor**. Em vez de procurar os valores de um dicionário por índices de números inteiros, você deve procurá-los por chaves, que

podem ser números ou strings. Podemos criar um dicionário usando o símbolo de chaves ({}) e separando os pares **chave:valor** com vírgulas. O screenshot a seguir é um exemplo de como criaríamos um dicionário com contagens de frutas – ele exibe o número de maçãs e, em seguida, adiciona um novo tipo de fruta e sua contagem:

```
>>> example_dict = {'apples':5, 'oranges':8}
>>> example_dict['apples']
5
>>> example_dict['bananas'] = 13
>>> example_dict
{'apples': 5, 'oranges': 8, 'bananas': 13}
>>>
```

Figura 1.2 – Exemplo de dicionário.

Há várias outras características que distinguem a linguagem Python e só queremos mostrar algumas aqui, sem entrar em muitos detalhes. Na verdade, provavelmente você usará pacotes como o **pandas** (pandas) e o **NumPy** (numpy) para quase toda a manipulação de dados que fará em Python. O NumPy fornece cálculo numérico rápido com arrays e matrizes, enquanto o pandas oferece vários recursos de preparação e exploração de dados com tabelas chamadas **DataFrames**. No entanto, é bom estar familiarizado com alguns aspectos básicos do Python – a linguagem que dá suporte a tudo isso. Por exemplo, a indexação funciona no NumPy e no pandas da mesma maneira que funciona em Python.

Um dos pontos fortes do Python é que ele é open source e tem uma comunidade de desenvolvedores ativa criando ferramentas extraordinárias. Usaremos várias dessas ferramentas neste livro. Um possível problema do uso de pacotes open source de diferentes colaboradores são as dependências entre os diversos pacotes. Por exemplo, se você quiser instalar o pandas, ele pode depender de uma versão específica do NumPy, que pode ou não ter sido instalada. Os sistemas de gerenciamento de pacotes facilitam a vida nesse aspecto. Quando você instalar um novo pacote por intermédio do sistema de gerenciamento de pacotes, ele assegurará que todas as dependências sejam atendidas. Se não forem, você será solicitado a fazer um upgrade ou a instalar novos pacotes conforme necessário.

Neste livro, usaremos o sistema de gerenciamento de pacotes **Anaconda**, que você precisa instalar. Aqui só utilizaremos Python, mas também é possível executar R com o Anaconda.

Nota: Ambientes – Se você já instalou e está usando o Anaconda, é

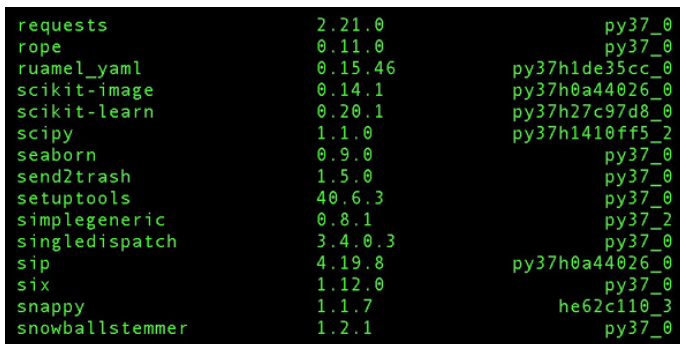
recomendável criar um novo ambiente Python 3.x para o livro. Os ambientes são como instalações separadas, em que o conjunto de pacotes instalado pode ser diferente, assim como a versão do Python. Eles são úteis para o desenvolvimento de projetos que precisem ser implantados em diferentes versões da linguagem. Para obter mais informações, consulte <https://conda.io/docs/user-guide/tasks/manage-environments.html>.

Exercício 1: Examinando o Anaconda e familiarizando-se com o Python

Neste exercício, você examinará os pacotes de sua instalação do Anaconda e manipulará um fluxo de controle básico e algumas estruturas de dados do Python, inclusive um loop for, e as estruturas dict e list. Isso confirmará que concluiu as etapas de instalação do prefácio e mostrará como a sintaxe e as estruturas de dados do Python podem ser um pouco diferentes das de outras linguagens de programação que você conhece. Execute as etapas a seguir para fazer o exercício:

Nota: O arquivo do código deste exercício pode ser encontrado aqui: <http://bit.ly/2Oyag1h>.

1. Abra um Terminal, se estiver usando o macOS ou o Linux, ou uma janela de prompt de comando no Windows. Digite `conda list` na linha de comando. Você deve ver uma saída semelhante à seguinte:



```
requests          2.21.0          py37_0
rope              0.11.0          py37_0
ruamel_yaml      0.15.46         py37h1de35cc_0
scikit-image     0.14.1          py37h0a44026_0
scikit-learn     0.20.1          py37h27c97d8_0
scipy            1.1.0           py37h1410ff5_2
seaborn          0.9.0           py37_0
send2trash       1.5.0           py37_0
setuptools       40.6.3          py37_0
simplegeneric     0.8.1           py37_2
singledispatch   3.4.0.3         py37_0
sip              4.19.8          py37h0a44026_0
six              1.12.0          py37_0
snappy           1.1.7           he62c110_3
snowballstemmer  1.2.1           py37_0
```

Figura 1.3 – Seleção de pacotes de conda list.

É possível ver todos os pacotes que estão instalados em seu ambiente. Veja quantos pacotes já vêm com a instalação padrão do Anaconda! Estão incluídos todos os pacotes necessários para a leitura do livro. No entanto, é fácil e descomplicado instalar novos pacotes e atualizar os existentes com o Anaconda; essa é uma das

principais vantagens de um sistema de gerenciamento de pacotes.

2. Digite python no Terminal para abrir um interpretador Python de linha de comando. Você deve obter uma saída semelhante à seguinte:

```
Python 3.7.1 (default, Dec 14 2018, 13:28:58)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figura 1.4 – Linha de comando Python.

Você verá algumas informações sobre sua versão do Python, assim como o prompt de comando da linguagem (>>>). Ao digitar algo depois do prompt, estará escrevendo código Python.

Nota: Usaremos o Jupyter Notebook neste livro, mas um dos objetivos do exercício é percorrermos as etapas básicas de criação e execução de programas Python no prompt de comando.

3. Crie um loop for no prompt de comando para exibir os valores de 0 a 4 usando o código a seguir:

```
for counter in range(5):
...   print(counter)
...
```

Quando você pressionar *Enter* após ver (...) no prompt, deve obter esta saída:

```
0
1
2
3
4
>>> █
```

Figura 1.5 – Saída de um loop for na linha de comando.

Observe que, em Python, a abertura do loop for é seguida por dois pontos e o **corpo do loop requer recuo**. É normal o uso de quatro espaços no recuo de um bloco de código. Aqui, o loop for está exibindo os valores retornados pelo iterador range(), que os acessou repetidamente usando a variável counter com a palavra-chave in.

Nota: Para ver mais detalhes sobre as convenções dos códigos Python, acesse a página a seguir: <https://www.python.org/dev/peps/pep-0008/>.

Agora, retornaremos ao nosso exemplo de dicionário. A primeira

etapa é criar o dicionário.

4. Crie um dicionário de frutas (apples [maças], oranges [laranjas] e bananas [bananas]) usando o código a seguir:

```
example_dict = {'apples':5, 'oranges':8, 'bananas':13}
```

5. Converta o dicionário em uma lista usando a função list(), como mostrado neste fragmento de código:

```
dict_to_list = list(example_dict)
dict_to_list
```

Quando você executar o código anterior, deve ver a seguinte saída:

```
['apples', 'oranges', 'bananas']
```

Figura 1.6 – Chaves do dicionário convertidas em uma lista.

Observe que, quando essa operação é executada, ao examinarmos o conteúdo, vemos que só as chaves do dicionário foram capturadas. Se quiséssemos os valores, teríamos de especificar isso com o método .values() da lista. Observe também que as chaves da lista estão na mesma ordem em que foram escritas na criação do dicionário. No entanto, isso não é garantido, já que os dicionários são tipos de coleção não ordenada.

Algo conveniente que você pode fazer com as listas é acrescentar outras listas a elas com o operador +. Como exemplo, na próxima etapa combinaremos a lista de frutas existente com uma lista que só contém mais um tipo de fruta, sobrepondo a variável que armazena a lista original.

6. Use o operador + para combinar a lista de frutas existente com uma nova lista contendo apenas uma fruta (pears [peras]):

```
dict_to_list = dict_to_list + ['pears']
dict_to_list
```

```
['apples', 'oranges', 'bananas', 'pears']
```

Figura 1.7 – Aumentando uma lista.

E se quiséssemos classificar nossa lista de tipos de frutas?

O Python fornece uma função sorted() interna que pode ser usada para esse fim; ela retornará uma versão classificada da entrada. Em nosso caso, isso significa que a lista de tipos de frutas será classificada alfabeticamente.

7. Classifique a lista de frutas em ordem alfabética usando a função

sorted(), como mostrado no fragmento a seguir:

```
sorted(dict_to_list)
```

Quando você executar o código anterior, deve ver a seguinte saída:

```
['apples', 'bananas', 'oranges', 'pears']
```

Figura 1.8 – Classificando uma lista.

Por enquanto, já vimos o suficiente de Python. Vamos mostrar-lhe como executar o código deste livro, logo, seu conhecimento de Python deve melhorar ao avançarmos.

Nota: À medida que você aprender mais e inevitavelmente quiser fazer novas experiências, é recomendável consultar a documentação: <https://docs.python.org/3/>.

Diferentes tipos de problemas da ciência de dados

Provavelmente você passará grande parte de seu tempo como cientista de dados preparando dados: descobrindo como obtê-los, obtendo-os, examinando-os, verificando se estão corretos e completos e associando-os a outros tipos de dados. O pandas facilitará esse processo. No entanto, se pretende ser um cientista de dados da área de machine learning, terá de dominar a arte e a ciência da **modelagem preditiva**. Isso significa usar um modelo matemático, ou uma formulação matemática idealizada, para aprender que relacionamentos estão ocorrendo dentro dos dados, na esperança de fazer previsões precisas e úteis quando entrarem novos dados.

Para esse fim, normalmente os dados são organizados em uma estrutura tabular, com **características** e uma **variável de resposta**. Por exemplo, se você quisesse prever o preço de uma casa com base em algumas características dela, como a **área** e o **número de quartos**, esses atributos seriam as características, e o **preço da casa** seria a variável de resposta, também chamada de **variável alvo** ou **variável dependente**, enquanto as características podem ser chamadas de **variáveis independentes**.

Se você tivesse um dataset de 1.000 casas incluindo os valores das características e os preços das casas, poderia dizer que possui 1.000 **amostras** de dados **rotulados**, em que os rótulos são os valores conhecidos da variável de resposta: os preços das diferentes casas.

Normalmente, a estrutura de dados tabular é organizada de modo que linhas diferentes são amostras distintas, enquanto as características e a resposta ocupam colunas diferentes, junto com outros metadados como os IDs das amostras, conforme mostrado na Figura 1.9.

	Número de Quartos			
3000000				
3000000				
3000000				

Figura 1.9 – Dados rotulados (os preços das casas são a variável alvo conhecida).

Problema de regressão

Uma vez que você tiver treinado um modelo para aprender o relacionamento entre as características e a resposta usando os dados rotulados, poderá empregá-lo para fazer previsões para casas das quais não souber o preço, com base nas informações contidas nas características. O objetivo da modelagem preditiva nesse caso é fazermos uma previsão que fique próxima do valor real da casa. Já que estamos prevendo um valor numérico em uma escala contínua, isso se chama **problema de regressão**.

Problema de classificação

Por outro lado, se estivéssemos tentando fazer uma previsão qualitativa da casa, para responder a uma pergunta de resposta **sim** ou **não** como “essa casa estará à venda nos próximos cinco anos?” ou “o proprietário não precisará pagar a hipoteca?”, estaríamos resolvendo o que é conhecido como **problema de classificação**. Nesse caso, esperamos dar a resposta afirmativa ou negativa certa. A Figura 1.10 é um diagrama que ilustra como o treinamento de modelos funciona e quais podem ser os resultados dos modelos de regressão ou classificação:

Fase de treinamento do modelo

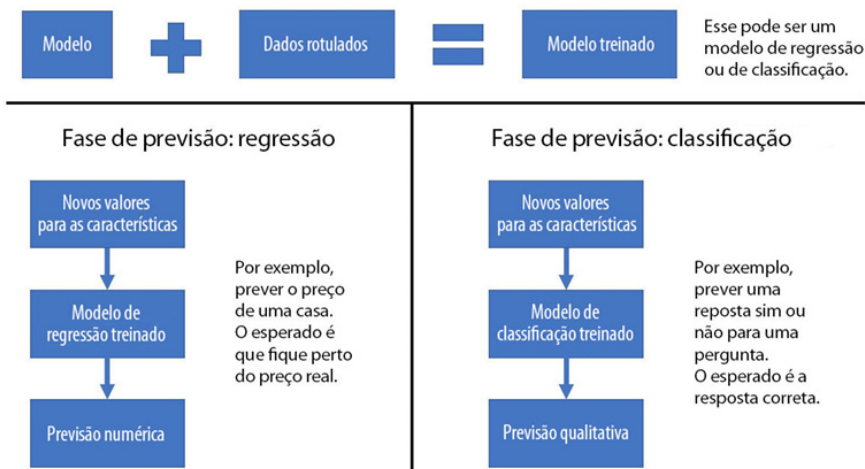


Figura 1.10 – Diagrama de treinamento e previsão de um modelo para a regressão e a classificação.

As tarefas de classificação e regressão são chamadas de **aprendizado supervisionado**, um tipo de problema que depende de dados rotulados. São problemas que demandam a “inspeção” dos valores conhecidos da variável alvo. Por outro lado, também há o **aprendizado não supervisionado**, que está relacionado a perguntas menos limitadas para as quais tentamos encontrar algum tipo de estrutura em um dataset que nem sempre tem rótulos. De um modo geral, qualquer tipo de problema de matemática aplicada, inclusive de áreas tão variadas quanto **otimização**, **inferência estatística** e **modelagem de séries temporais**, pode ser considerado como uma responsabilidade apropriada para um cientista de dados.

Carregando os dados do estudo de caso com o Jupyter e o pandas

É hora de darmos uma primeira olhada nos dados que usaremos em nosso estudo de caso. A única coisa que faremos nesta seção é verificar se conseguimos carregar os dados corretamente em um **Jupyter Notebook**. As tarefas de examinar os dados e entender o problema que resolveremos com eles virão depois.

O arquivo de dados é uma planilha do Excel chamada `default_of_credit_card_clients_courseware_version_1_13_19.xls`.

Recomendamos que você abra assim que puder a planilha no Excel ou

no programa de planilhas de sua escolha. Observe o número de linhas e colunas e examine alguns exemplos de valores. Isso o ajudará a saber se conseguiu ou não a carregar corretamente no Jupyter Notebook.

Nota: O dataset pode ser obtido no link a seguir: <http://bit.ly/2HIk5t3>. Essa é uma versão modificada do dataset original, que extraímos do UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: Universidade da Califórnia, School of Information and Computer Science.

O que é um Jupyter Notebook?

Os Jupyter Notebooks são ambientes de codificação interativos que permitem o uso de gráficos e texto inline. São ótimas ferramentas para os cientistas de dados divulgarem e preservarem seus resultados, já que tanto os métodos (código) quanto a mensagem (texto e gráficos) são integrados. O ambiente pode ser considerado como um tipo de página web em que podemos escrever e executar códigos. Na verdade, os Jupyter Notebooks podem ser renderizados como páginas web, e isso ocorre no GitHub. Você pode ver um de nossos exemplos de notebooks em <http://bit.ly/2OvndJg>. Examine-o para ter uma ideia do que pode ser feito. Um trecho desse notebook aparece na Figura 1.11, exibindo código, elementos gráficos e texto, que nesse contexto é chamado de **markdown**.

Uma das coisas que é preciso aprender sobre os Jupyter Notebooks é como navegar e fazer edições. Há dois modos disponíveis. Se você selecionar uma célula e pressionar *Enter*, estará no **modo de edição** e poderá editar o texto dessa célula. Se pressionar *Esc*, estará no **modo de comando** e poderá navegar pelo notebook.

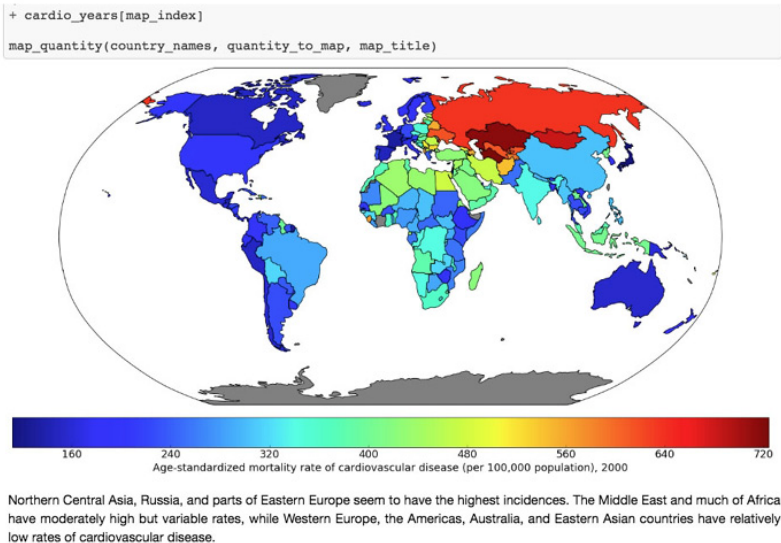


Figura 1.11 – Exemplo de um Jupyter Notebook exibindo código, elementos gráficos e texto markdown.

Quando estiver no modo de comando, poderá usar muitas hotkeys úteis. As setas *para cima* e *para baixo* o ajudarão a selecionar diferentes células e a rolar pelo notebook. Se você pressionar *y* em uma célula selecionada no modo de comando, ele a alterará para uma **célula de código**, na qual o texto é interpretado como código. Pressionar *m* a alterará para uma **célula markdown**. A combinação *Shift + Enter* avalia a célula, renderizando o markdown ou executando o código, conforme o caso.

Nossa primeira tarefa em nosso primeiro Jupyter Notebook será carregar os dados do estudo de caso. Para fazê-lo, usaremos uma ferramenta chamada **pandas**. Não é exagero dizer que o **pandas** é a ferramenta que se destaca na preparação de dados em Python.

Um **DataFrame** é uma classe básica do **pandas**. Falaremos mais sobre o que é uma classe posteriormente, mas você pode considerá-la como um template para uma estrutura de dados, em que a estrutura poderia ser algo como as listas ou os dicionários que já discutimos. No entanto, um **DataFrame** é muito mais rico em funcionalidade do que essas estruturas. Em muitos aspectos, ele é semelhante às planilhas. Há linhas, que são rotuladas por um índice de linha, e colunas, que em geral recebem rótulos semelhantes a cabeçalhos que podem ser considerados como o índice da coluna. Na verdade, **Index** é um tipo de dado do **pandas** usado para armazenar os índices de um **DataFrame**, e as colunas têm o próprio tipo de dado chamado **Series**.

Muitas das coisas que podemos fazer com as planilhas do Excel também podem ser feitas com um DataFrame, como criar tabelas dinâmicas e filtrar linhas. O pandas também inclui uma funcionalidade semelhante à do SQL. Você pode associar diferentes DataFrames, por exemplo. Outra vantagem dos DataFrames é que, uma vez que os dados estão contidos em um deles, temos os recursos de várias funcionalidades do pandas à nossa disposição. A Figura 1.12 é um exemplo de um DataFrame do pandas:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	b730d678-5446	20000	2	2	1	24	2
1	99a3ae70-57a8	120000	2	2	2	26	-1
2	90194006-f94a	90000	2	2	2	34	0
3	19acf9a3-2b01	50000	2	2	1	37	0
4	70d7bc16-a6a4	50000	1	2	1	57	-1

Figura 1.12 – Exemplo de um DataFrame do pandas com um índice de linha inteiro à esquerda e um índice de coluna na forma de strings.

Na verdade, o exemplo da Figura 1.12 são os dados do estudo de caso. Como primeira etapa no Jupyter e no pandas, veremos como criar um Jupyter Notebook e carregar dados com o pandas. Há várias funções convenientes que você pode usar no pandas para explorar seus dados, inclusive `.head()` para ver as primeiras linhas do DataFrame, `.info()` para ver todas as colunas com os tipos de dados, `.columns` para retornar uma lista de nomes de colunas como strings, e outras que aprenderemos nos próximos exercícios.

Exercício 2: Carregando os dados do estudo de caso em um Jupyter Notebook

Agora que você conhece os Jupyter Notebooks, o ambiente no qual escreveremos código, e o pandas, o pacote de preparação de dados, criaremos nosso primeiro Jupyter Notebook. Usaremos o pandas dentro desse notebook para carregar os dados do estudo de caso e os examinaremos brevemente. Execute as etapas a seguir para fazer o exercício:

Nota: Para os Exercícios 2–5 e a Atividade 1, o código e a saída resultante foram carregados em um Jupyter Notebook que pode

ser encontrado em <http://bit.ly/2W9cwPH>. Você pode rolar para a seção apropriada dentro do Jupyter Notebook a fim de localizar o exercício ou a atividade que deseja.

1. Abra um Terminal (macOS ou Linux) ou uma janela de prompt de comando (Windows) e digite `jupyter notebook`.

Você verá a interface do Jupyter em seu navegador web. Se ele não abrir automaticamente, copie e cole a URL do terminal para o navegador. Nessa interface, você poderá navegar em seus diretórios começando por aquele em que estava quando iniciou o servidor do notebook.

2. Navegue para um local conveniente para armazenar os materiais deste livro e crie um novo notebook Python 3 a partir do menu **New**, como mostrado na Figura 1.13:



Figura 1.13 – Tela inicial do Jupyter.

3. Faça com que sua primeira célula seja do tipo markdown digitando `m` no modo de comando (pressione `Esc` para entrar no modo de comando) e digite o símbolo de número, `#`, no começo da primeira linha, seguido por um espaço, para o cabeçalho. Crie um título para seu notebook aqui. Nas próximas linhas, insira uma descrição.

Na Figura 1.14 temos um screenshot de um exemplo, incluindo outros tipos de markdown como negrito, itálico e a maneira de escrever texto no estilo código em uma célula markdown:

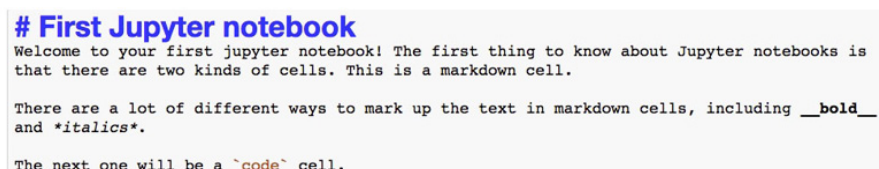


Figura 1.14 – Célula markdown não renderizada.

Observe que é boa prática criar um título e uma breve descrição

para o notebook, a fim de mostrar sua finalidade para os leitores.

4. Pressione *Shift* + *Enter* para renderizar a célula markdown.

Isso também deve criar uma nova célula, que será uma célula de código. Você pode alterá-la para uma célula markdown, já que agora sabe como fazê-lo, pressionando *m*, e transformá-la novamente em uma célula de código pressionando *y*. Sabemos que é uma célula de código porque estamos vendo `In [ ]:` próximo a ela.

5. Digite `import pandas as pd` na nova célula, como mostrado no screenshot a seguir:

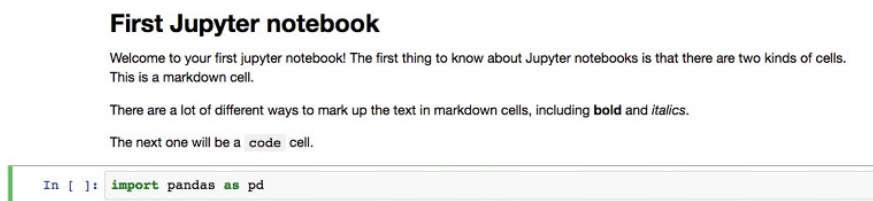


Figura 1.15 – Célula markdown e célula de código renderizadas.

Quando você executar essa célula, a biblioteca pandas será carregada em seu ambiente de computação. É comum a importação com “as” para que seja criado um alias para a biblioteca. Agora usaremos o pandas para carregar o arquivo de dados. Ele está no formato do Microsoft Excel, logo, podemos usar `pd.read_excel`.

Nota: Para obter mais informações sobre todas as opções possíveis para `pd.read_excel`, consulte a documentação a seguir: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_excel.html.

6. Importe o dataset, que está no formato do Excel, como um DataFrame usando o método `pd.read_excel()`, como mostrado neste fragmento:

```
df = pd.read_excel('../Data/default_of_credit_card_clients_courseware_version_1_21_19.xls')
```

Observe que você precisa apontar o leitor do Excel para o local onde o arquivo está localizado. Se ele estiver no mesmo diretório de seu notebook, é possível inserir apenas o nome do arquivo. O método `pd.read_excel` carregará o arquivo do Excel em um DataFrame, que chamamos de `df`. Agora o poder do pandas está disponível para nós.

Faremos algumas verificações rápidas nas próximas etapas. Em primeiro lugar, os números de linhas e colunas coincidem com o

que vemos ao examinar o arquivo no Excel?

7. Use o método `.shape` para examinar o número de linhas e colunas, como mostrado na linha de código a seguir:

```
df.shape
```

Quando você executar a célula, verá a seguinte saída:

```
In [3]: df.shape
Out[3]: (30000, 25)
```

Figura 1.16 – Verificando a forma de um DataFrame.

Ela deve coincidir com o visto na planilha. Caso contrário, será preciso inspecionar as diversas opções de `pd.read_excel` para ver se é necessário ajustar algo.

Com esse exercício, carregamos com sucesso nosso dataset no Jupyter Notebook. Você também pode examinar os métodos `.info()` e `.head()`, que fornecerão informações sobre todas as colunas e mostrarão as primeiras linhas do DataFrame, respectivamente. Agora você já está com seus dados no pandas.

Como observação final, embora já deva ter ficado claro, é bom ressaltar que, se você definir uma variável em uma única célula de código, ela ficará disponível em outras células de código dentro do notebook. As células de código de um notebook compartilham o escopo contanto que o kernel esteja sendo executado, como mostrado no screenshot a seguir:

```
In [4]: a = 5
In [5]: a
Out[5]: 5
```

Figura 1.17 – Variável em escopo entre células.

Familiarizando-se com os dados e executando sua limpeza

Suponhamos que estivéssemos examinando os dados pela primeira vez. No trabalho como cientista de dados, há vários cenários a partir dos quais você poderia receber o dataset, por exemplo:

1. Você criou a consulta SQL que gerou os dados.
2. Um colega criou uma consulta SQL para você, com sua

participação.

3. Um colega que sabe sobre os dados os forneceu, sem que você pedisse.
4. Você recebeu um dataset sobre o qual pouco sabe.

Nos casos 1 e 2, você participou da geração/extração dos dados. Nesses cenários, entendeu o problema da empresa e encontrou os dados necessários com a ajuda de um engenheiro de dados ou fez sua própria pesquisa e projetou a consulta SQL que os gerou. Com frequência, principalmente quando ganhamos mais experiência na função de cientista de dados, a primeira etapa é reunir-se com o sócio da empresa para entender e aperfeiçoar a definição matemática do problema. Em seguida, você desempenharia um papel-chave na definição do conteúdo do dataset.

Mesmo se você tiver um nível relativamente alto de familiaridade com os dados, sua exploração e a verificação de **sínteses estatísticas** de diferentes variáveis ainda será uma importante primeira etapa. Essa etapa o ajudará a selecionar boas características ou lhe dará ideias de como gerar características novas. No entanto, no terceiro e no quarto casos, dos quais você não participou ou tinha pouco conhecimento sobre os dados, a exploração é ainda mais importante.

Outra etapa inicial importante do processo de ciência de dados é examinar o **dicionário de dados**, o qual, como o termo sugere, é um documento que explica o que o proprietário dos dados acha que deve ser incluído no conjunto, como as definições dos rótulos das colunas. É função do cientista de dados percorrer o conjunto com cuidado para verificar se essas impressões coincidem com o que de fato existe nos dados. Nos casos 1 e 2, provavelmente você terá de criar por conta própria o dicionário de dados, que deve ser considerado documentação essencial do projeto. Nos casos 3 e 4, deve tentar obter o dicionário se possível.

Os dados de estudo de caso que usaremos neste livro são basicamente semelhantes aos do caso 3.

O problema da empresa

Nosso cliente é uma empresa de cartão de crédito. Eles nos trouxeram um dataset que inclui dados demográficos e dados financeiros recentes (últimos seis meses) de uma amostra de 30.000 titulares de contas. Esses dados estão no nível de conta de crédito; em outras palavras, há

uma linha para cada conta (você deve sempre esclarecer qual é a definição de linha, em um dataset). As linhas são rotuladas de acordo com se no mês seguinte ao período de dados histórico de seis meses um proprietário de conta ficou inadimplente, ou seja, não fez o pagamento mínimo.

Objetivo

Seu objetivo é desenvolver um modelo que preveja se uma conta ficará inadimplente no próximo mês, de acordo com dados demográficos e históricos. Posteriormente no livro, discutiremos a aplicação prática do modelo.

Os dados já estão preparados e um dicionário de dados está disponível. O dataset fornecido com o livro, `default_of_credit_card_clients_courseware_version_1_21_19.xls`, é uma versão modificada do dataset do UCI Machine Learning Repository: https://archive.ics.uci.edu/ml/datasets/default_of_credit_card_clients. Examine essa página web, que inclui o dicionário de dados.

Nota: O dataset original foi obtido no UCI Machine Learning Repository [<http://archive.ics.uci.edu/ml>]. Irvine, CA: Universidade da Califórnia, School of Information and Computer Science. Neste livro, modificamos o dataset para que atenda a nossos objetivos. O dataset modificado pode ser encontrado aqui: <http://bit.ly/2HIk5t3>.

Etapas da exploração de dados

Agora que conhecemos o problema da empresa e temos uma ideia do que os dados devem conter, podemos comparar essas impressões com o que estamos vendo realmente nos dados. Nossa função ao explorar dados é percorrê-los tanto diretamente quanto usando resumos numéricos e gráficos e considerar criticamente se fazem sentido e correspondem ao que nos foi dito sobre eles. Algumas etapas úteis para a exploração de dados são:

1. Saber quantas colunas os dados contêm.
Podem ser de características, resposta ou metadados.
2. Quantas linhas (amostras).
3. Que tipos de características existem. Quais são **categóricas** e quais são **numéricas**?

Os valores das características categóricas pertencem a classes

discretas como “sim”, “não” ou “talvez”. Normalmente as características numéricas pertencem a uma escala numérica contínua, como as quantias em dólares.

4. Qual é a aparência dos dados segundo essas características.

Para saber isso, você pode examinar o intervalo de valores em características numéricas ou a frequência de classes diferentes em características categóricas, por exemplo.

5. Há dados faltando?

Já respondemos às perguntas 1 e 2 da seção anterior; há 30.000 linhas e 25 colunas. Para responder ao resto das perguntas no próximo exercício, usaremos a ferramenta pandas como guia. Começaremos verificando a integridade básica dos dados no exercício a seguir.

Nota: Se compararmos a descrição do dicionário de dados existente no site, observe que X6–X11 chamam-se PAY_1–PAY_6 em nossos dados. Da mesma forma, X12–X17 passam a ser BILL_AMT1–BILL_AMT6 e X18–X23 são PAY_AMT1–PAY_AMT6.

Exercício 3: Verificando a integridade básica dos dados

Neste exercício, executaremos uma verificação básica para saber se o dataset contém o que esperamos e examinaremos se há o número correto de amostras.

Os dados devem ter observações referentes a 30.000 contas de crédito. Embora haja 30.000 linhas, também devemos verificar se existem 30.000 IDs de contas exclusivos. É possível que, se a consulta SQL usada para gerar os dados foi executada com um esquema desconhecido, os valores que deveriam ser exclusivos na verdade não o sejam.

Para examinar isso, podemos verificar se o número de IDs de contas exclusivos é igual ao número de linhas. Execute as etapas a seguir para fazer o exercício:

Nota: O código e o gráfico de saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Examine os nomes das colunas executando o comando a seguir na célula:

```
df.columns
```

O método `.columns` do `DataFrame` está sendo empregado para examinarmos os nomes de todas as colunas. Você obterá a saída a seguir quando executar a célula:

```
df.columns
```

```
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_1',  
      'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',  
      'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',  
      'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',  
      'default payment next month'],  
      dtype='object')
```

Figura 1.18 – Colunas do dataset.

Como podemos ver, os nomes de todas as colunas estão listados na saída. A coluna de IDs de contas chama-se `ID`. As outras colunas parecem ser as características, com a última coluna sendo a variável de resposta. Examinaremos resumidamente as informações do dataset que nos foi dado pelo cliente:

LIMIT_BAL: Valor do crédito fornecido (em novos dólares taiwaneses (NT)) inclusive o crédito do consumidor individual e familiar (complementar).

SEX: Gênero (1 = masculino; 2 = feminino).

Nota: Não usaremos os dados de gênero para tomar decisões de solvibilidade devido a considerações éticas.

EDUCATION: Instrução (1 = pós-graduação; 2 = universidade; 3 = ensino médio; 4 = outros).

MARRIAGE: Estado civil (1 = casado; 2 = solteiro; 3 = outros).

AGE: Idade (ano).

PAY_1–Pay_6: Registro de pagamentos passados. Pagamentos mensais passados, registrados de abril a setembro, são armazenados nessas colunas.

PAY_1 representa o status de reembolso em setembro; **PAY_2** = status de reembolso em agosto; e assim por diante até **PAY_6**, que representa o status de reembolso em abril.

A escala de medida do status de reembolso é a seguinte: -1 = pagamento pontual; 1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento; e assim por diante até 8 = atraso de oito meses no pagamento; 9 = atraso de nove meses ou mais no pagamento.

BILL_AMT1–BILL_AMT6: Valor da fatura (em novos dólares

taiwaneses).

BILL_AMT1 representa o valor da fatura em setembro; BILL_AMT2 representa o valor da fatura em agosto; e assim por diante até BILL_AMT7, que representa o valor da fatura em abril.

PAY_AMT1–PAY_AMT6: Valor de pagamentos anteriores (novos dólares taiwaneses). PAY_AMT1 representa o valor pago em setembro; PAY_AMT2 representa o valor pago em agosto; e assim por diante até PAY_AMT6, que representa o valor pago em abril.

Usaremos o método `.head()` na próxima etapa para observar as primeiras linhas de dados.

2. Digite o comando a seguir na célula subsequente e execute-o usando *Shift + Enter*:

```
df.head()
```

Você verá a seguinte saída:

```
df.head()
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1
0	798fc410-45c1	20000	2	2	1	24	2
1	8a8c8f3b-8eb4	120000	2	2	2	26	-1
2	85698822-43f5	90000	2	2	2	34	0
3	0737c11b-be42	50000	2	2	1	37	0
4	3b7f77cc-dbc0	50000	1	2	1	57	-1

5 rows × 25 columns

Figura 1.19 – Método `.head()` de um `DataFrame`.

A coluna ID parece conter identificadores exclusivos. Para verificar se isso ocorre realmente em todo o dataset, podemos contar o número de valores exclusivos usando o método `.nunique()` na série (ou seja, coluna) ID. Primeiro, selecionaremos a coluna usando colchetes.

3. Selecione a coluna alvo (ID) e conte os valores exclusivos usando o comando a seguir:

```
df['ID'].nunique()
```

É possível ver nessa saída que temos 29.687 entradas exclusivas:

```
df['ID'].nunique()
```

```
29687
```

Figura 1.20 – Descobrimos um problema de qualidade de dados.

4. Execute o comando a seguir para obter o número de linhas do dataset:

```
df.shape
```

Como podemos ver na saída, temos um total de 30.000 linhas no dataset:

```
df.shape
```

```
(30000, 25)
```

Figura 1.21 – Dimensões do dataset.

Identificamos que o número de IDs exclusivos é menor do que o número de linhas. Isso significa que o ID não é um identificador exclusivo para as linhas de dados, como pensávamos. Sabemos então que há alguma duplicação de IDs. No entanto, em que quantidade? Um único ID está sendo duplicado várias vezes? Quantos IDs estão sendo duplicados?

Podemos usar o método `.value_counts()` na série ID para começar a responder a essas perguntas. Ele é semelhante a um procedimento **group by/count** em SQL e listará os IDs exclusivos e a frequência com que ocorrem. Executaremos essa operação na próxima etapa e armazenaremos as contagens de valores em uma variável `id_counts`.

5. Armazene as contagens de valores em uma variável definida como `id_counts` e exiba os valores armazenados usando o método `.head()`, como mostrado:

```
id_counts = df['ID'].value_counts()
```

```
id_counts.head()
```

Você obterá a seguinte saída:

```
id_counts = df['ID'].value_counts()
id_counts.head()
```

```
e50d8395-da32    2
4534975d-bf92    2
a3a5c0fc-fdd6    2
0d66d575-c461    2
fd6033f4-cc72    2
Name: ID, dtype: int64
```

Figura 1.22 – Obtendo contagens de valores de IDs de contas.

Observe que `.head()` retorna as cinco primeiras linhas por padrão.

Você pode especificar o número de itens a serem exibidos passando o número desejado nos parênteses, ().

- Exiba o número de entradas duplicadas agrupadas executando outra contagem de valores:

```
id_counts.value_counts()
```

Obteremos esta saída:

```
id_counts.value_counts()
1      29374
2       313
Name: ID, dtype: int64
```

Figura 1.23 – Obtendo contagens de valores das IDs de contas.

Na saída anterior e na contagem de valores inicial, podemos ver que a maioria dos IDs ocorre exatamente uma única vez, como esperado. No entanto, 313 IDs ocorrem duas vezes. Logo, nenhum ID ocorre mais do que duas vezes. De posse dessas informações, estamos prontos para começar a examinar mais detalhadamente esse problema de qualidade de dados e corrigi-lo. Criaremos máscaras booleanas para limpar melhor os dados.

Máscaras booleanas

Para ajudar a limpar os dados do estudo de caso, introduziremos o conceito de **máscara lógica**, também conhecida como **máscara booleana**. Uma máscara lógica é uma maneira de filtrar um array, ou série, obedecendo alguma condição. Por exemplo, podemos usar o operador “igual a” em Python, `=`, para encontrar todos os locais de um array que contêm um valor específico. Outras comparações, como “maior que” (`>`), “menor que” (`<`), “maior ou igual a” (`>=`) e “menor ou igual a” (`<=`), também podem ser usadas. A saída dessa comparação é um array com uma série de valores True/False, também conhecidos como valores **booleanos**. Cada elemento da saída corresponderá a um elemento da entrada, com o resultado True se a condição for atendida; caso contrário, obteremos False. Para ilustrar como funciona, usaremos **dados sintéticos**, ou seja, dados criados para a exploração ou a ilustração de um conceito. Primeiro, importaremos o pacote NumPy, que tem muitos recursos de geração de números aleatórios, e daremos a ele o alias `np`:

```
import numpy as np
```

Agora usaremos o que é chamado de **seed** (semente) no gerador de

números aleatórios. Se você definir um seed, obterá os mesmos resultados entre as execuções do gerador. Caso contrário, isso não é garantido. Essa pode ser uma opção útil se você usa números aleatórios em seu trabalho e quer ter resultados consistentes sempre que executar um notebook:

```
np.random.seed(seed = 24)
```

Em seguida, geraremos 100 inteiros aleatórios, selecionados entre 1 e 5 (inclusive). Para fazê-lo, podemos usar `numpy.random.randint`, com os argumentos apropriados.

```
random_integers = np.random.randint(low = 1, high = 5, size = 100)
```

Examinaremos os cinco primeiros elementos desse array, com `random_integers[:5]`. A saída deve ser a seguinte:

```
array([3, 4, 1, 4, 2])
```

Figura 1.24 – Inteiros aleatórios.

Suponhamos que quiséssemos conhecer os locais de todos os elementos de `random_integers` igual a 3. Poderíamos criar uma máscara booleana para fazer isso.

```
is_equal_to_3 = random_integers == 3
```

Pela verificação dos primeiros cinco elementos, sabemos que só o primeiro elemento é igual a 3, mas não os outros. Logo, em nossa máscara booleana, esperamos `True` na primeira posição e `False` nas 4 posições seguintes. É isso que ocorre?

```
is_equal_to_3[:5]
```

O código anterior deve fornecer esta saída:

```
array([ True, False, False, False, False])
```

Figura 1.25 – Máscara booleana para os números inteiros.

É isso que esperávamos. Esse exemplo mostra a criação de uma máscara booleana. No entanto, o que mais podemos fazer? Suponhamos que quiséssemos saber quantos elementos são iguais a 3. Nesse caso, você pode fazer a soma de uma máscara booleana, que interpreta `True` como 1 e `False` como 0:

```
sum(is_equal_to_3)
```

Esse código deve fornecer a saída a seguir:

Faz sentido, já que, com uma seleção aleatória igualmente provável de 5 valores possíveis, o esperado é que cada valor apareça 20% das vezes. Além de ver quantos valores do array atendem à condição booleana, também podemos usar a máscara booleana para selecionar os elementos do array original que atendem a essa condição. As máscaras booleanas podem ser usadas diretamente para indexar arrays, como mostrado aqui:

```
random_integers[is_equal_to_3]
```

Essa instrução exibe os elementos de `random_integers` que atendem à condição booleana que especificamos. Nesse caso, os 22 elementos iguais a 3:

```
array([3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])
```

Figura 1.27 – Usando a máscara booleana para indexar um array.

Agora você conhece os aspectos básicos dos arrays booleanos, que são úteis em muitas situações. Especificamente, você pode usar o método `.loc` de DataFrames para indexar suas linhas por uma máscara booleana e as colunas por rótulo. Continuaremos a explorar os dados do estudo de caso com essas habilidades.

Exercício 4: Continuando a verificação da integridade dos dados

Nesse exercício, com nosso conhecimento dos arrays booleanos, examinaremos alguns dos IDs duplicados que descobrimos. No Exercício 3, verificamos que nenhum ID apareça mais de duas vezes. Podemos usar essa informação para localizar os IDs duplicados e examiná-los. Em seguida, tomaremos medidas para remover linhas de qualidade duvidosa do dataset. Execute as etapas a seguir para fazer o exercício:

Nota: O código e o gráfico da saída desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Continuando onde paramos no Exercício 3, queremos os índices da série `id_counts` cuja contagem é 2 para localizar as duplicatas. Atribuiremos os IDs duplicados a uma variável chamada `dupe_mask` e exibiremos os cinco primeiros IDs duplicados usando os comandos a seguir:

```
dupe_mask = id_counts == 2  
dupe_mask[0:5]
```

Você obterá a seguinte saída:

```
47d9ee33-0df0    True
db903e22-a55a    True
9878723a-0b58    True
8d3a2576-a958    True
956cbf4a-d24e    True
Name: ID, dtype: bool
```

Figura 1.28 – Máscara booleana para localizar IDs duplicados.

Aqui, `dupe_mask` é a máscara lógica que criamos para armazenar os valores booleanos.

Observe que, na saída anterior, estamos exibindo só as cinco primeiras entradas usando `dupe_mask` para ilustrar o conteúdo desse array. Como sempre, você pode editar os índices nos colchetes (`[]`) para alterar o número de entradas exibidas.

Nossa próxima etapa é usar a máscara lógica para selecionar os IDs que estão duplicados. Os próprios IDs foram incluídos como índice da série `id_count`. Podemos acessar o índice para usar nossa máscara lógica para fins de seleção.

2. Acesse o índice de `id_count` e exiba as cinco primeiras linhas como contexto usando o comando a seguir:

```
id_counts.index[0:5]
```

Com essa instrução, você obterá a seguinte saída:

```
Index(['47d9ee33-0df0', 'db903e22-a55a', '9878723a-0b58', '8d3a2576-a958',
      '956cbf4a-d24e'],
      dtype='object')
```

Figura 1.29 – IDs duplicados.

3. Selecione e armazene os IDs duplicados em uma nova variável chamada `dupe_ids` usando o comando a seguir:

```
dupe_ids = id_counts.index[dupe_mask]
```

4. Converta `dupe_ids` em uma lista e obtenha seu tamanho usando os seguintes comandos:

```
dupe_ids = list(dupe_ids)
len(dupe_ids)
```

Você deve ver esta saída:

Figura 1.30 – Saída exibindo o tamanho da lista.

Alteramos a variável `dupe_ids` para uma lista, já que precisaremos dela nessa forma em etapas futuras. A lista tem um tamanho igual a

313, como pode ser visto na saída anterior, que coincide com o número de IDs duplicados que obtivemos na contagem de valores.

5. Verificaremos os dados de `dupe_ids` exibindo as cinco primeiras entradas com o comando a seguir:

```
dupe_ids[0:5]
```

A seguinte saída é obtida:

```
dupe_ids[0:5]
```

```
[ '47d9ee33-0df0',  
  'db903e22-a55a',  
  '9878723a-0b58',  
  '8d3a2576-a958',  
  '956cbf4a-d24e' ]
```

Figura 1.31 – Criando uma lista de IDs duplicados.

Podemos observar na saída anterior que a lista contém as entradas de IDs duplicados requeridas. Estamos prontos para examinar os dados dos IDs de nossa lista de duplicatas. Especificamente, queremos examinar os valores das características, para ver o que, se houver algo, há de diferente entre essas entradas duplicadas. Usaremos os métodos `.isin` e `.loc` para esse fim.

Usando os três primeiros IDs de nossa lista de duplicatas, `dupe_ids[0:3]`, primeiro queremos encontrar as linhas que contêm esses IDs. Se passarmos essa lista de IDs para o método `.isin` da série `ID`, ele criará outra máscara lógica que poderemos usar no `DataFrame` maior para exibir as linhas que contêm os IDs. O método `.isin` ficará aninhado em uma instrução `.loc` de indexação do `DataFrame` para a seleção do local de todas as linhas que contêm “True” na máscara booleana. O segundo argumento da instrução de indexação `.loc` é `:`, que implica que todas as colunas serão selecionadas. Ao executar as próximas etapas, estaremos basicamente filtrando o `DataFrame` para visualizar todas as colunas dos três primeiros IDs duplicados.

6. Execute o comando a seguir em seu Notebook para pôr em prática o plano que formulamos na etapa anterior:

```
df.loc[df['ID'].isin(dupe_ids[0:3]),:].head(10)
```

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_1	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5
11769	9878723a-0b58	130000	1	5	1	44	0	0	0	0	...	54668	22780
11869	9878723a-0b58	0	0	0	0	0	0	0	0	0	...	0	0
14979	db903e22-a55a	50000	1	2	2	55	2	0	0	0	...	15848	16026
15079	db903e22-a55a	0	0	0	0	0	0	0	0	0	...	0	0
23377	47d9ee33-0df0	550000	2	2	1	32	2	0	0	0	...	548020	530672
23477	47d9ee33-0df0	0	0	0	0	0	0	0	0	0	...	0	0

6 rows x 25 columns

Figura 1.32 – Examinando os dados de IDs duplicados.

O que estamos vendo aqui é que cada ID duplicado parece ter uma linha de dados válidos e outra somente com zeros. Pare um momento e pense o que poderia fazer com essa informação.

Após alguma reflexão, deve ter ficado claro que você deve excluir as linhas só de zeros. Elas podem ter surgido devido a uma condição de associação errada na consulta SQL que gerou os dados. De qualquer forma, uma linha só de zeros definitivamente contém dados inválidos e não faria sentido alguém ter 0 anos, limite de crédito igual a 0 e assim por diante.

Uma abordagem para lidarmos com esse problema seria encontrar as linhas que só têm zeros, exceto na primeira coluna, que tem os IDs. Esses dados seriam inválidos, e, se os excluíssemos, também resolveríamos nosso problema de IDs duplicados. Podemos encontrar as entradas do DataFrame que são iguais a zero criando uma matriz booleana com o mesmo tamanho do DataFrame inteiro, usando a condição “é igual a zero”.

7. Crie uma matriz booleana com o mesmo tamanho do DataFrame inteiro usando `==`, como mostrado:

```
df_zero_mask = df == 0
```

Nas próximas etapas, usaremos `df_zero_mask`, que é outro DataFrame, contendo valores booleanos. O objetivo é criarmos uma série booleana, `feature_zero_mask`, que identifique cada linha em que todos os elementos a partir da segunda coluna (as características e a resposta, mas não os IDs) sejam 0. Para fazê-lo, primeiro temos de indexar `df_zero_mask` usando o método de indexação de inteiros (`.iloc`). Nesse método, passaremos (`:`) para examinar todas as linhas e (`1:`) para examinar todas as colunas a partir da segunda (índice 1). Para concluir, aplicaremos o método `all()` ao longo do eixo da coluna (`axis=1`) e ele retornará `True`

somente se todas as colunas dessa linha forem iguais a True. Mesmo sendo muita coisa para considerar, é fácil de codificar, como veremos na etapa a seguir.

8. Crie a série booleana `feature_zero_mask`, como mostrado aqui:

```
feature_zero_mask = df_zero_mask.iloc[:,1:].all(axis=1)
```

9. Calcule a soma da série booleana usando este comando:

```
sum(feature_zero_mask)
```

Você deve obter a seguinte saída:

315

Figura 1.33 – Número de linhas somente com zeros exceto pelo ID.

A saída anterior nos diz que 315 linhas têm zeros para cada coluna exceto a primeira. Esse número é maior do que o número de IDs duplicados (313), logo, se excluirmos todas as “linhas de zeros”, podemos nos livrar do problema dos IDs duplicados.

10. Limpe o DataFrame eliminando as linhas só com zeros, exceto pelo ID, usando o código a seguir:

```
df_clean_1 = df.loc[~feature_zero_mask,:].copy()
```

Ao executar a operação de limpeza da etapa anterior, retornamos um novo DataFrame chamado `df_clean_1`. Observe que usamos o método `.copy()` após a operação de indexação `.loc` para criar uma cópia dessa saída, em vez de visualizarmos o DataFrame original. Podemos considerar essa ação como a criação de um novo DataFrame em vez de referenciarmos o original. Dentro do método `.loc`, usamos o operador lógico not, `~`, para selecionar todas as linhas que não têm zeros para todas as características e a resposta, e: para selecionar todas as colunas. Esses são os dados válidos que queremos manter. Após fazer isso, queremos saber se o número de linhas restantes é igual ao número de IDs exclusivos.

11. Verifique o número de linhas e colunas de `df_clean_1` executando o código a seguir:

```
df_clean_1.shape
```

Você verá esta saída:

```
df_clean_1 = df.loc[~feature_zero_mask,:].copy()

df_clean_1.shape

(29685, 25)
```

12. Obtenha o número de IDs exclusivos executando este código:

```
df_clean_1['ID'].nunique()
```

```
df_clean_1['ID'].nunique()
```

```
29685
```

Figura 1.35 – Número de IDs exclusivos no DataFrame limpo.

Na saída anterior, podemos ver que eliminamos com sucesso as duplicatas, já que o número de IDs exclusivos é igual ao número de linhas. Relaxe e anime-se. Essa foi uma introdução complexa a algumas técnicas do pandas para indexação e caracterização de dados. Agora que extraímos os IDs duplicados, estamos prontos para começar a examinar os dados propriamente ditos: as características e a resposta. Vamos conduzi-lo por esse processo.

Exercício 5: Explorando e limpando os dados

Até agora, identificamos um problema de qualidade de dados relacionado aos metadados: fomos informados de que cada amostra de nosso dataset corresponderia a um ID de conta exclusivo, o que não ocorreu. Conseguimos usar a indexação lógica e o pandas para resolver o problema. Esse foi um problema básico de qualidade de dados, referente apenas a que amostras estavam presentes, com base nos metadados. Fora isso, na verdade não estamos interessados na coluna de metadados dos IDs de contas: por enquanto eles não nos ajudarão a desenvolver um modelo preditivo de inadimplência.

Estamos prontos para começar a examinar os valores das características e da resposta, os dados que usaremos para desenvolver nosso modelo preditivo. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Obtenha o tipo de dado das colunas do dataset usando o método `.info()` como mostrado:

```
df_clean_1.info()
```

Você deve ver a seguinte saída:

```
df_clean_1.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 29685 entries, 0 to 29999
Data columns (total 25 columns):
ID                29685 non-null object
LIMIT_BAL        29685 non-null int64
SEX              29685 non-null int64
EDUCATION        29685 non-null int64
MARRIAGE         29685 non-null int64
AGE              29685 non-null int64
PAY_1            29685 non-null object
PAY_2            29685 non-null int64
PAY_3            29685 non-null int64
PAY_4            29685 non-null int64
PAY_5            29685 non-null int64
PAY_6            29685 non-null int64
BILL_AMT1        29685 non-null int64
BILL_AMT2        29685 non-null int64
BILL_AMT3        29685 non-null int64
BILL_AMT4        29685 non-null int64
BILL_AMT5        29685 non-null int64
BILL_AMT6        29685 non-null int64
PAY_AMT1         29685 non-null int64
PAY_AMT2         29685 non-null int64
PAY_AMT3         29685 non-null int64
PAY_AMT4         29685 non-null int64
PAY_AMT5         29685 non-null int64
PAY_AMT6         29685 non-null int64
default payment next month 29685 non-null int64
dtypes: int64(23), object(2)
memory usage: 5.9+ MB
```

Figura 1.36 – Obtendo metadados das colunas.

Podemos ver na *Figura 1.34* que há 25 colunas. Cada linha tem 29.685 valores **não nulos**, de acordo com esse resumo, que é o número de linhas do DataFrame. Isso indicaria que não há dados ausentes, já que cada célula contém algum valor. No entanto, se houver um valor de preenchimento para representar dados ausentes, ele não ficaria evidente aqui.

Também vemos que a maioria das colunas exibe int64 próximo a elas, indicando que têm o tipo de dado **integer**, isto é, números como ..., -2, -1, 0, 1, 2,... . As exceções são ID e PAY_1. Já estamos familiarizados com ID; ela contém strings, que são IDs de contas. E quanto a PAY_1? De acordo com os valores do dicionário de dados, é esperado que contenha inteiros, como todas as outras características. Vejamos essa coluna com mais detalhes.

2. Use o método `.head(n)` do pandas para visualizar as `n` linhas superiores da série `PAY_1`:

```
df_clean_1['PAY_1'].head(5)
```

Você deve ver esta saída:

```
df_clean_1['PAY_1'].head(5)

0      2
1     -1
2      0
3      0
4     -1
Name: PAY_1, dtype: object
```

Figura 1.37 – Examine o conteúdo de algumas colunas.

Os inteiros da esquerda da saída são o índice, que são simplesmente inteiros consecutivos começando em 0. Os dados da coluna PAY_1 são mostrados à direita. Eles são o status de pagamento da fatura mensal mais recente, usando os valores -1, 1, 2, 3 e assim por diante. No entanto, podemos ver que há valores 0 aqui, que não estão documentados no dicionário de dados. De acordo com o dicionário de dados, “A escala de medida do status de reembolso é: -1 = pagamento pontual; 1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento; . . . ; 8 = atraso de oito meses no pagamento; 9 = atraso de nove meses ou mais no pagamento” (<https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients>). Faremos um exame mais detalhado, usando as contagens de valores dessa coluna.

3. Obtenha as contagens de valores da coluna PAY_1 usando o método `.value_counts()`:

```
df_clean1['PAY_1'].value_counts()
```

Você deve ver a saída a seguir:

```
df_clean_1['PAY_1'].value_counts()

0      13087
-1      5047
1       3261
Not available  3021
-2       2476
2       2378
3        292
4         63
5         23
8         17
6         11
7          9
Name: PAY_1, dtype: int64
```

Figura 1.38 – Contagens de valores da coluna PAY_1.

A saída anterior revela a presença de dois valores não documentados, 0 e -2, e também a razão de essa coluna ter sido importada pelo pandas como um tipo de dado object, em vez de int64 como era de se esperar para dados inteiros. Há uma string

'Not available' presente na coluna, simbolizando dados ausentes. Posteriormente no livro, retornaremos a ela quando considerarmos como lidar com dados ausentes. Por enquanto, removeremos as linhas do dataset nas quais a característica tem um valor ausente.

4. Use uma máscara lógica com o operador `!=` (que significa “diferente de” em Python) para encontrar todas as linhas que não têm dados ausentes para a característica de `PAY_1`:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'  
valid_pay_1_mask[0:5]
```

Ao executar o código anterior, você obterá a saída a seguir:

```
valid_pay_1_mask = df_clean_1['PAY_1'] != 'Not available'  
  
valid_pay_1_mask[0:5]  
  
0    True  
1    True  
2    True  
3    True  
4    True  
Name: PAY_1, dtype: bool
```

Figura 1.39 – Criando uma máscara booleana.

5. Verifique quantas linhas não têm dados ausentes calculando a soma da máscara:

```
sum(valid_pay_1_mask)
```

Você obterá esta saída:

```
sum(valid_pay_1_mask)  
  
26664
```

Figura 1.40 – Soma da máscara booleana para dados não ausentes.

Vemos que 26.664 linhas não têm o valor 'Not available' na coluna `PAY_1`. Na contagem de valores, 3.021 linhas tinham esse valor, e, já que $29.685 - 3.021 = 26.664$, isso está correto.

6. Limpe os dados eliminando as linhas de `PAY_1` com valores ausentes como mostrado aqui:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask,:].copy()
```

7. Obtenha a dimensão dos dados limpos usando o comando a seguir:

```
df_clean_2.shape
```

Você verá esta saída:

```
df_clean_2 = df_clean_1.loc[valid_pay_1_mask,:].copy()

df_clean_2.shape

(26664, 25)
```

Figura 1.41 – Dimensão dos dados limpos.

Após remover essas linhas, vemos que o DataFrame resultante tem a dimensão esperada. Você também pode verificar por sua própria conta se as contagens de valores indicam que os valores desejados foram removidos dessa forma: `df_clean_2['PAY_1'].value_counts()`.

Por fim, para que o tipo de dado dessa coluna seja consistente com os outros, vamos convertê-lo do tipo genérico object para int64 como em todas as outras características, usando o método `.astype`. Em seguida, selecionaremos duas colunas, inclusive PAY_1, para examinar os tipos de dados e nos certificarmos se funcionou.

8. Execute o comando a seguir para converter o tipo de dado de PAY_1 de object para int64 e exiba os metadados das colunas PAY_1 e PAY_2:

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')
df_clean_2[['PAY_1', 'PAY_2']].info()
```

```
df_clean_2['PAY_1'] = df_clean_2['PAY_1'].astype('int64')

df_clean_2[['PAY_1', 'PAY_2']].info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 26664 entries, 0 to 29999
Data columns (total 2 columns):
PAY_1      26664 non-null int64
PAY_2      26664 non-null int64
dtypes: int64(2)
memory usage: 624.9 KB
```

Figura 1.42 – Verifique o tipo de dado da coluna limpa.

Parabéns, você concluiu sua segunda operação de limpeza de dados! No entanto, deve se lembrar de que durante o processo também notamos os valores não documentados -2 e 0 em PAY_1. Suponhamos que entrássemos em contato novamente com o sócio da empresa e nos fosse dada a seguinte informação:

- -2 significa que a conta começou o mês sem valor a ser pago e o crédito não foi usado
- -1 significa que a conta usou um valor que foi totalmente pago
- 0 significa que o pagamento mínimo foi feito, mas o saldo total

devedor não foi pago (isto é, uma parcela do saldo devedor foi transportada para o próximo mês)

Agradecemos ao sócio da empresa, já que por enquanto essas informações esclarecem nossas dúvidas. É importante manter um canal de comunicação e uma relação de trabalho satisfatórios com o sócio da empresa, como você pode ver aqui, porque isso é capaz de determinar o sucesso ou a falha de um projeto.

Exploração e garantia da qualidade dos dados

Até aqui, resolvemos dois problemas de qualidade dos dados apenas fazendo perguntas básicas ou examinando o resumo de `.info()`. Examinaremos agora as primeiras colunas. Antes de chegarmos no histórico de pagamento de faturas, temos os limites de crédito das contas em `LIMIT_BAL` e as características demográficas `SEX`, `EDUCATION`, `MARRIAGE` e `AGE`. O sócio da empresa nos procurou para informar que o gênero não deve ser usado na previsão de solvibilidade, já que por seus padrões seria **antiético**. Logo, memorizaremos isso para referência futura. Exploraremos então o resto dessas colunas, fazendo as correções necessárias.

Para explorar melhor os dados, usaremos **histogramas**. Os histogramas são uma boa maneira de visualizar dados que estejam em uma escala contínua, como valores monetários e faixas etárias. Um histograma agrupa valores semelhantes em bins e exibe o número de pontos de dados existentes nesses bins como um gráfico de barras.

Para plotar os histogramas, nos familiarizaremos com os recursos gráficos do pandas, que conta com uma biblioteca chamada **Matplotlib** para criar gráficos, logo, também definiremos algumas opções usando o `matplotlib`. Empregando essas ferramentas, aprenderemos como obter rapidamente sínteses estatísticas dos dados no pandas.

Exercício 6: Explorando o limite de crédito e as características demográficas

Nesse exercício, começaremos nossa exploração dos dados com o limite de crédito e as características etárias. Vamos visualizá-los e obter sínteses estatísticas para verificar se os dados contidos nessas características são aceitáveis. Em seguida, examinaremos as características categóricas de instrução e estado civil para ver se os

valores fazem sentido e os corrigiremos se necessário. LIMIT_BAL e AGE são características numéricas, o que significa que são medidas em uma escala contínua. Consequentemente, usaremos histogramas para visualizá-las. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Importe o matplotlib e defina algumas opções de plotagem com este fragmento de código:

```
import matplotlib.pyplot as plt #importa o pacote de plotagem
```

```
#renderiza a plotagem automaticamente
```

```
%matplotlib inline
```

```
import matplotlib as mpl #recurso adicional de plotagem
```

```
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
```

Esse código importa o matplotlib e usa `.rcParams` para definir a resolução (dpi = pontos por polegada) para obtenção de uma imagem com boa nitidez; não é preciso se preocupar com essa parte a não ser que você esteja preparando material para apresentação, já que isso pode tornar as imagens grandes demais em seu notebook.

2. Execute `df_clean_2[['LIMIT_BAL', 'AGE']].hist()` e deve ver os histogramas a seguir:

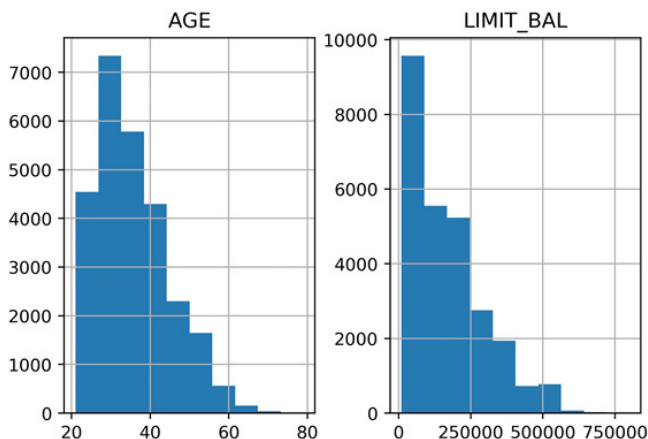


Figura 1.43 – Histogramas dos dados de limite de crédito e idade.

Esse é um bom snapshot visual dessas características. Podemos obter uma rápida visualização aproximada de todos os dados dessa forma. Para vermos sínteses estatísticas, como a média e a mediana

(isto é, o quinquagésimo percentil), há outra função útil do pandas.

3. Gere um relatório tabular de síntese estatística usando o comando a seguir:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

Você deve ver esta saída:

```
df_clean_2[['LIMIT_BAL', 'AGE']].describe()
```

	LIMIT_BAL	AGE
count	26664.000000	26664.000000
mean	167919.054905	35.505213
std	129839.453081	9.227442
min	10000.000000	21.000000
25%	50000.000000	28.000000
50%	140000.000000	34.000000
75%	240000.000000	41.000000
max	800000.000000	79.000000

Figura 1.44 – Sínteses estatísticas de dados de limite de crédito e idade.

Com base nos histogramas e nas convenientes estatísticas calculadas por `.describe()`, que inclui uma contagem de não nulos, a média e o desvio-padrão, o valor mínimo, o valor máximo e os quartis, podemos chegar a algumas conclusões.

LIMIT_BAL, o limite de crédito, parece fazer sentido. Os limites de crédito têm um valor mínimo igual a 10.000. Esse dataset é de Taiwan; a unidade monetária (novos dólares taiwaneses) não é familiar, mas intuitivamente sabemos que o limite de crédito deve ser um valor acima de zero. Recomendamos que você tente converter para a moeda local e considere esses limites. Por exemplo, 1 dólar americano é igual a cerca de 30 novos dólares taiwaneses.

A característica AGE também parece bem distribuída, com ninguém com idade abaixo de 21 anos possuindo uma conta de crédito.

Para as características categóricas, é útil verificar as contagens de valores, já que há relativamente poucos valores exclusivos.

4. Obtenha as contagens de valores da característica EDUCATION usando o código a seguir:

```
df_clean_2['EDUCATION'].value_counts()
```

Você deve ver esta saída:

```
df_clean_2['EDUCATION'].value_counts()

2      12458
1       9412
3       4380
5        245
4        115
6         43
0         11
Name: EDUCATION, dtype: int64
```

Figura 1.45 – Contagens de valores da característica EDUCATION.

Aqui, vemos os graus de instrução não documentados 0, 5 e 6, já que o dicionário de dados descreve apenas “Instrução (1 = pós-graduação; 2 = universidade; 3 = ensino médio; 4 = outros)”. O sócio da empresa nos disse que não conhece os outros graus. Já que eles não são predominantes, vamos agrupá-los na categoria “outros”, que parece apropriada, claro que com o consentimento de nosso cliente.

5. Execute este código para combinar os graus não documentados da característica EDUCATION com o grau “outros” e examine os resultados:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)
df_clean_2['EDUCATION'].value_counts()
```

O método `.replace` do pandas ajuda a fazermos rapidamente as substituições descritas na etapa anterior. Quando você executar o código, deve ver esta saída:

```
df_clean_2['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)

df_clean_2['EDUCATION'].value_counts()

2      12458
1       9412
3       4380
4        414
Name: EDUCATION, dtype: int64
```

Figura 1.46 – Limpando a característica EDUCATION.

Observe que fizemos essa alteração **in loco** (`inplace=True`). Isso significa que, em vez de retornar um novo DataFrame, a operação fará a alteração no DataFrame existente.

6. Obtenha as contagens de valores da característica MARRIAGE usando o código a seguir:

```
df_clean_2['MARRIAGE'].value_counts()
```

Você deve ver a seguinte saída:

```
df_clean_2['MARRIAGE'].value_counts()

2      14158
1      12172
3        286
0         48
Name: MARRIAGE, dtype: int64
```

Figura 1.47 – Contagens de valores da característica MARRIAGE bruta.

O problema aqui é semelhante ao encontrado na característica EDUCATION; há um valor, 0, que não está documentado no dicionário de dados: “1 = casado; 2 = solteiro; 3 = outros”. Logo, vamos agrupá-lo com “outros”.

7. Altere os valores 0 da característica MARRIAGE para 3 e examine o resultado com este código:

```
df_clean_2['MARRIAGE'].replace(to_replace=0, value=3, inplace=True)
df_clean_2['MARRIAGE'].value_counts()
```

A saída deve ser:

```
2      14158
1      12172
3        334
Name: MARRIAGE, dtype: int64
```

Figura 1.48 – Contagens de valores da característica MARRIAGE limpa.

Fizemos uma extensa exploração e limpeza dos dados. Posteriormente, executaremos operações mais avançadas de visualização e exploração das características de histórico financeiro, que vêm depois disso no DataFrame.

Aprofundamento nas características categóricas

Os algoritmos de machine learning só funcionam com números. Se seus dados contiverem características textuais, por exemplo, você terá de transformá-las em números. Na verdade, já sabemos que os dados de nosso estudo de caso são totalmente numéricos. No entanto, é interessante ponderar por que precisa ser assim. Especificamente, considere a característica EDUCATION.

Esse é um exemplo do que é conhecido como **característica categórica**: sabemos que, como dados brutos, essa coluna seria composta dos rótulos de texto ‘pós-graduação’, ‘universidade’, ‘ensino médio’ e ‘outros’. Esses são os chamados **níveis** da característica categórica; aqui, há quatro níveis. Somente por intermédio de um mapeamento, que já foi definido para nós, é que esses dados existem

como os números 1, 2, 3 e 4 em nosso dataset. Essa atribuição específica de categorias a números cria o que é conhecido como **característica ordinal**, já que os níveis são mapeados para números em ordem. Como cientista de dados, você precisa conhecer esses mapeamentos, se não foi quem os escolheu.

Quais são as implicações desse mapeamento?

Faz sentido que os graus de instrução sejam categorizados, com 1 correspondendo ao mais alto grau de nosso dataset, 2 ao grau superior seguinte, 3 ao próximo e 4 presumivelmente incluindo os graus mais baixos. No entanto, quando usarmos essa codificação como característica numérica em um modelo de machine learning, ela será tratada como qualquer outra característica numérica. Para alguns modelos, esse efeito pode não ser o desejado.

E se a função de um modelo for encontrar um relacionamento linear entre as características e a resposta?

Se isso vai funcionar ou não dependerá do relacionamento real entre os diferentes graus de instrução e o resultado que estivermos tentando prever.

Examinaremos dois casos hipotéticos de variáveis categóricas ordinais, com 10 níveis em cada um. Os níveis medem os graus de satisfação relatados por clientes que visitam um site. O período médio em minutos gasto no site pelos clientes que relataram cada nível está plotado no eixo y. Também plotamos a linha de melhor ajuste em cada caso para ilustrar como um modelo linear lidaria com esses dados, como mostrado na Figura 1.49:

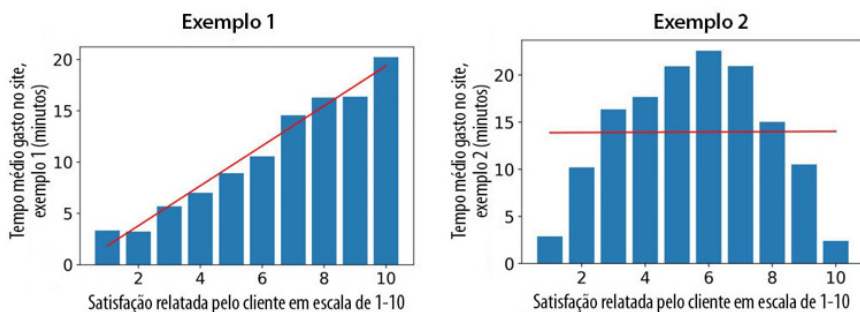


Figura 1.49 – Características ordinais podem ou não funcionar bem em um modelo linear.

Podemos ver que, se um algoritmo assumir um relacionamento linear

(linha reta) entre as características e a resposta, isso pode ou não funcionar bem dependendo do relacionamento real entre essa característica e a variável de resposta. Observe que, no exemplo anterior, estamos modelando um problema de regressão: a variável de resposta adota um intervalo de números contínuo. No entanto, alguns algoritmos de classificação como a **regressão logística** também assumem um efeito linear das características. Discutiremos isso com mais detalhes posteriormente quando modelarmos os dados de nosso estudo de caso.

De um modo geral, para um modelo de classificação binária, você pode examinar os diferentes níveis de uma característica categórica em relação aos valores médios da variável de resposta, os quais representam as “taxas” da classe positiva (isto é, as amostras em que a variável de resposta = 1) para cada nível. Isso pode lhe dar uma ideia de se uma codificação ordinal funcionará bem com um modelo linear. Supondo que você tenha importado para o seu Jupyter notebook os mesmos pacotes das seções anteriores, poderá verificar isso rapidamente usando groupby/agg e a plotagem de barras no pandas:

```
f_clean_2.groupby('EDUCATION').agg({'default payment next  
month': 'mean'}).plot.bar(legend=False)  
plt.ylabel('Default rate')  
plt.xlabel('Education level: ordinal encoding')
```

Quando você executar o código, deve ver a saída a seguir:

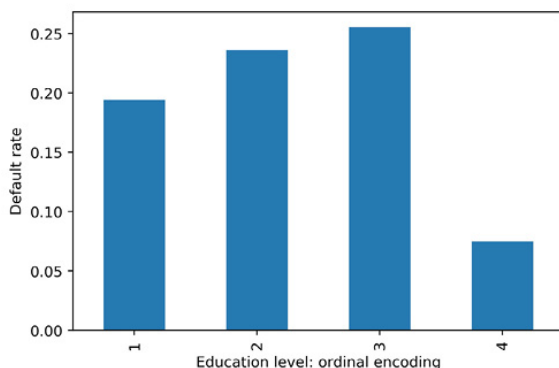


Figura 1.50 – Taxa de inadimplência dentro dos graus de instrução.

Como no exemplo 2 da *Figura 1.49*, aqui parece que um ajuste de linha reta não seria a melhor descrição dos dados. Caso uma característica tenha um efeito não linear como esse, pode ser melhor usar um algoritmo mais complexo como uma **árvore de decisão** ou **floresta aleatória**. Ou, se um modelo linear mais simples e

interpretável como a regressão logística for desejado, poderíamos evitar uma codificação ordinal e usar uma maneira diferente de codificar variáveis categóricas. Uma maneira popular de fazer isso se chama **codificação one-hot (OHE, one-hot encoding)**.

A OHE é uma maneira de transformarmos uma característica categórica, que pode ser composta de rótulos de texto nos dados brutos, em uma característica numérica que possa ser usada em modelos matemáticos.

Aprenderemos isso em um exercício. E, se você estiver se perguntando por que uma regressão logística é mais interpretável e uma floresta aleatória é mais complexa, aprenderemos esses conceitos com detalhes no decorrer do livro.

Nota: Variáveis categóricas em diferentes pacotes de machine learning

Alguns pacotes de machine learning, por exemplo, certos pacotes do R ou versões mais recentes da plataforma Spark para big data, manipulam variáveis categóricas sem assumir que elas sejam ordinais. Certifique-se sempre de ler cuidadosamente a documentação para saber o que o modelo assumirá sobre as características, e como especificar se uma variável é categórica, se essa opção estiver disponível.

Exercício 7: Implementando a OHE para uma característica categórica

Nesse exercício, executaremos uma “engenharia reversa” na característica EDUCATION do dataset para obter os rótulos de texto que representam os diferentes graus de instrução e mostraremos como usar o pandas para criar uma OHE.

Primeiro, consideraremos nossa característica EDUCATION, antes de ela ser codificada como um ordinal. Pelo dicionário de dados, sabemos que 1 = pós-graduação (graduate school), 2 = universidade (university), 3 = ensino médio (high school), 4 = outros (others). Queremos criar uma coluna que tenha essas strings, em vez de números. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2W9cwPH>.

1. Crie uma coluna vazia para os rótulos categóricos chamada EDUCATION_CAT usando o comando a seguir:

```
df_clean_2['EDUCATION_CAT'] = 'none'
```

2. Examine as primeiras linhas do DataFrame referentes às colunas EDUCATION e EDUCATION_CAT:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

A saída deve ser esta:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	none
1	2	none
2	2	none
3	2	none
4	2	none
5	1	none
6	1	none
7	2	none
8	3	none
9	3	none

Figura 1.51 – Selecionando colunas e visualizando as 10 primeiras linhas.

Precisamos preencher essa nova coluna com as strings apropriadas. O pandas fornece uma funcionalidade conveniente para o mapeamento dos valores de uma série (Series) para novos valores. Na verdade, essa função chama-se `.map` e usa um dicionário para estabelecer a correspondência entre os valores antigos e os novos. Nosso objetivo aqui é mapear os números de EDUCATION para as strings que eles representam. Por exemplo, onde a coluna EDUCATION for igual a 1, atribuiremos a string 'graduate school' à coluna EDUCATION_CAT, e assim por diante para os outros graus de instrução.

3. Crie um dicionário que descreva o mapeamento das categorias de instrução usando o código a seguir:

```
cat_mapping = {  
    1: "graduate school",  
    2: "university",  
    3: "high school",  
    4: "others"  
}
```

4. Aplique o mapeamento à coluna EDUCATION original usando `.map` e atribua o resultado à nova coluna EDUCATION_CAT:

```
df_clean_2['EDUCATION_CAT'] = df_clean_2['EDUCATION'].map(cat_mapping)
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

Após executar essas linhas, você deve ver esta saída:

```
df_clean_2[['EDUCATION', 'EDUCATION_CAT']].head(10)
```

	EDUCATION	EDUCATION_CAT
0	2	university
1	2	university
2	2	university
3	2	university
4	2	university
5	1	graduate school
6	1	graduate school
7	2	university
8	3	high school
9	3	high school

Figura 1.52 – Examinando os valores de string correspondentes à codificação ordinal de EDUCATION.

Excelente! Observe que poderíamos ter pulado a *Etapa 1*, na qual atribuímos a nova coluna com 'none', e ter ido direto para as *Etapas 3 e 4* para criar a nova coluna. No entanto, às vezes é útil criar uma nova coluna inicializada com um único valor, logo, é bom saber como fazê-lo.

Estamos prontos para a codificação one-hot. Podemos executá-la passando uma série (Series) de um DataFrame para a função `get_dummies()` do pandas. A função recebeu esse nome porque as colunas de codificação one-hot também são chamadas de **variáveis dummy**. O resultado será um novo DataFrame, com um número igual de colunas e níveis da variável categórica.

5. Execute esse código para criar um DataFrame de codificação one-hot da coluna EDUCATION_CAT. Examine as 10 primeiras linhas:

```
edu_ohe = pd.get_dummies(df_clean_2['EDUCATION_CAT'])
edu_ohe.head(10)
```

O código deve produzir a saída a seguir:

```
edu_oh = pd.get_dummies(df_clean_2['EDUCATION_CAT'])
edu_oh.head(10)
```

	graduate school	high school	none	others	university
0	0	0	0	0	1
1	0	0	0	0	1
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	1	0	0	0	0
6	1	0	0	0	0
7	0	0	0	0	1
8	0	1	0	0	0
9	0	1	0	0	0

Figura 1.53 – Data frame de codificação one-hot.

Agora podemos ver por que essa abordagem chama-se “codificação one-hot”: em todas as colunas, qualquer linha específica terá um número 1 em exatamente 1 coluna e zeros nas outras. Em uma linha específica, a coluna com o 1 deve corresponder ao nível da variável categórica original. Para verificar isso, precisamos concatenar esse novo DataFrame com o original e examinar os resultados lado a lado. Usaremos a função `concat` do pandas, para a qual passaremos a lista de DataFrames que queremos concatenar e a palavra-chave `axis=1` solicitando que eles sejam concatenados horizontalmente, isto é, ao longo do eixo da coluna. Basicamente, isso significa que estamos combinando esses dois DataFrames “lado a lado”, o que sabemos que podemos fazer porque acabamos de criar esse novo DataFrame a partir do original: sabemos que ele terá o mesmo número de linhas, que estarão na mesma ordem do DataFrame original.

6. Concatene o DataFrame de codificação one-hot com o original da seguinte forma:

```
df_with_oh = pd.concat([df_clean_2, edu_oh], axis=1)
df_with_oh[['EDUCATION_CAT', 'graduate school',
            'high school', 'university', 'others']].head(10)
```

Você deve ver esta saída:

```
df_with_ohe = pd.concat([df_clean_2, edu_ohe], axis=1)
df_with_ohe[['EDUCATION_CAT', 'graduate school',
              'high school', 'university', 'others']].head(10)
```

	EDUCATION_CAT	graduate school	high school	university	others
0	university	0	0	1	0
1	university	0	0	1	0
2	university	0	0	1	0
3	university	0	0	1	0
4	university	0	0	1	0
5	graduate school	1	0	0	0
6	graduate school	1	0	0	0
7	university	0	0	1	0
8	high school	0	1	0	0
9	high school	0	1	0	0

Figura 1.54 – Verificando as colunas de codificação one-hot.

Certo, parece que funcionou como esperado. A OHE é outra maneira de codificar características categóricas que evita a estrutura numérica inerente a uma codificação ordinal. No entanto, observe o que ocorreu aqui: pegamos uma única coluna, EDUCATION, e a explodimos em um número de colunas igual aos dos níveis da característica. Nesse caso, já que há apenas quatro níveis, não foi tão difícil. Porém, se sua variável categórica tivesse um número muito grande de níveis, seria melhor considerar uma estratégia alternativa, como agrupar alguns níveis em categorias individuais.

É um bom momento para salvarmos o DataFrame criado aqui, que resume nossos esforços de limpar os dados e adicionar uma coluna OHE.

Selecione um nome de arquivo e grave o último DataFrame em um arquivo CSV (de valor separado por vírgulas) desta forma: `df_with_ohe.to_csv('../Data/Chapter_1_cleaned_data.csv', index=False)`; não incluímos o índice, já que isso não é necessário e pode gerar colunas adicionais quando fizermos o carregamento posteriormente.

Explorando as características de histórico financeiro do dataset

Estamos prontos para explorar o resto das características do dataset do estudo de caso. Primeiro, tentaremos carregar um DataFrame a partir

do arquivo **CSV** que salvamos no fim da última seção. Isso pode ser feito com o uso do fragmento a seguir:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Observe que, se você continuar escrevendo código no mesmo notebook, isso sobreporá o valor contido anteriormente na variável `df`, que era o `DataFrame` de dados brutos. Recomendamos que verifique as funções `.head()`, `.columns` e `.shape` do `DataFrame`. São itens importantes a verificar sempre que carregamos um `DataFrame`. Não o faremos aqui por falta de espaço, mas fizemos no notebook que acompanha o livro.

Nota: O caminho de seu arquivo CSV pode ser diferente dependendo de onde você o salvou.

As características que faltam ser examinadas são as de histórico financeiro. Elas se encaixam naturalmente em três grupos: o status dos pagamentos mensais nos últimos seis meses e as quantias cobradas e pagas no mesmo período. Primeiro examinaremos o status dos pagamentos. É conveniente relacioná-los em uma lista para podermos estudá-los em conjunto.

```
pay_feats = ['PAY_1', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6']
```

Podemos usar o método `.describe` nessas seis séries para examinar sínteses estatísticas:

```
df[pay_feats].describe()
```

Isso deve produzir a saída a seguir:

```
df[pay_feats].describe()
```

	PAY_1	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6
count	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000
mean	-0.017777	-0.133363	-0.167679	-0.225023	-0.269764	-0.293579
std	1.126769	1.198640	1.199165	1.167897	1.131735	1.150229
min	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000	-2.000000
25%	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	8.000000	8.000000	8.000000	8.000000	8.000000	8.000000

Figura 1.55 – Síntese estatística das características de status de pagamento.

Aqui, podemos ver que o intervalo de valores é o mesmo para todas as características: -2, -1, 0, ... 8. Parece que o valor 9, descrito no

dicionário de dados como “atraso de nove meses ou mais no pagamento”, nunca ocorre.

Já explicamos o significado de todos esses valores, alguns dos quais não se encontram no dicionário de dados original. Agora examinaremos novamente as contagens de valores de PAY_1, dessa vez classificada pelos valores que estamos contando, que são o índice desta série:

```
df[pay_feats[0]].value_counts().sort_index()
```

Esse código deve produzir a seguinte saída:

```
df[pay_feats[0]].value_counts().sort_index()
-2      2476
-1      5047
 0     13087
 1      3261
 2      2378
 3       292
 4        63
 5        23
 6         11
 7          9
 8         17
Name: PAY_1, dtype: int64
```

Figura 1.56 – Contagens de valores do status dos pagamentos do mês anterior.

Em comparação com os valores inteiros positivos, a maioria dos valores são -2, -1 ou 0, que correspondem a uma conta que estava em boa situação no mês passado: não usada, totalmente paga ou com pelo menos o pagamento mínimo feito.

Observe que, devido à definição dos outros valores dessa variável (1 = atraso de um mês no pagamento; 2 = atraso de dois meses no pagamento e assim por diante), essa característica é como um híbrido de características categóricas e numéricas. Por que a não utilização do crédito deve corresponder a um valor igual a -2, enquanto o valor 2 significa atraso de 2 meses no pagamento etc.? Precisamos entender que a codificação numérica de status de pagamento -2, -1 e 0 constitui uma decisão tomada pelo criador do dataset sobre como codificar certas características categóricas, que seriam agrupadas em uma característica que na verdade é numérica: o número de meses de atraso no pagamento (valores igual a 1 e maiores). Posteriormente, consideraremos os possíveis efeitos dessa forma de fazer as coisas sobre a capacidade preditiva dessa característica.

Por enquanto, continuaremos simplesmente explorando os dados. Esse

dataset é suficientemente pequeno, com 18 características financeiras e algumas outras, para podermos examinar individualmente cada característica. Se ele tivesse milhares de características, provavelmente abandonaríamos essa abordagem e exploraríamos técnicas de **redução de dimensionalidade**, que são maneiras de condensar as informações de um grande número de características em um número menor de características derivadas, ou, alternativamente, métodos de **seleção de características**, que podem ser usados para isolar as características importantes a partir de um grupo com várias candidatas. Demonstraremos e explicaremos algumas técnicas de seleção de características posteriormente. Nesse dataset, é viável visualizar cada característica. Como vimos no último capítulo, um histograma é uma boa maneira de obtermos uma interpretação visual rápida do mesmo tipo de informações que obteríamos com tabelas de contagens de valores. Você pode usá-lo nas características de status de pagamento do último mês com `df[pay_feats[0]].hist()`, para produzir a saída mostrada na Figura 1.57.

Agora examinaremos com detalhes como esse gráfico foi produzido e consideraremos se ele é tão informativo quanto deveria. Um ponto-chave sobre a funcionalidade gráfica do pandas é que **na verdade a sua plotagem chama o matplotlib em segundo plano**. Observe que o último argumento disponível para o método `.hist()` do pandas é `**kwds`, que a documentação indica serem argumentos de palavra-chave do matplotlib.

Nota: Para obter mais informações, consulte a página a seguir:
<https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.hist.html>.

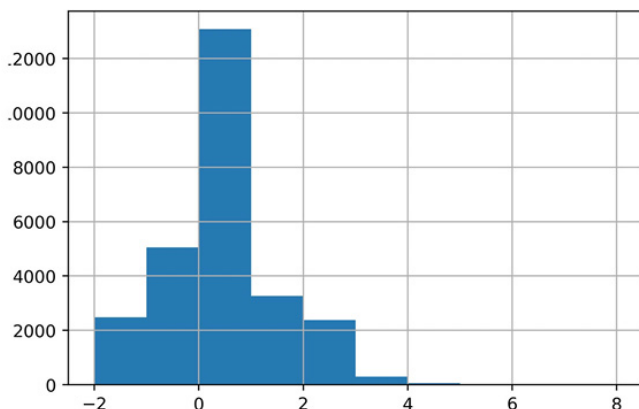


Figura 1.57 – Histograma de PAY_1 usando argumentos padrão.

Se examinarmos a documentação do matplotlib no que diz respeito a `matplotlib.pyplot.hist`, ela mostra argumentos adicionais que podemos usar com o método `.hist()` do pandas, como o tipo de histograma a ser plotado (consulte https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html para ver mais detalhes). Em geral, para a obtenção de mais detalhes sobre a funcionalidade de plotagem, é importante incluir o matplotlib, e, em alguns cenários, é melhor usá-lo diretamente, em vez do pandas, se você quiser ter mais controle sobre a aparência das plotagens.

É bom ressaltar que o pandas usa o matplotlib, que por sua vez usa o NumPy. Na plotagem de histogramas com o matplotlib, o cálculo numérico dos valores que compõem o histograma na verdade é executado pela função `.histogram` do NumPy. Esse é um exemplo típico de reutilização de código, ou de “não reinventar a roda”. Se uma funcionalidade padrão, como a plotagem de um histograma, já tem uma boa implementação em Python, não há razão para criar outra. E, se a operação matemática que cria os dados do histograma já foi implementada, ela também deve ser utilizada. Isso mostra a interconectividade do ecossistema Python.

Agora abordaremos algumas questões importantes que surgem no cálculo e na plotagem de histogramas.

Número de bins

Os histogramas funcionam agrupando valores no que chamamos de **bins**. O número de bins é o número de barras verticais que compõem a plotagem discreta do histograma que vemos. Se houver uma grande quantidade de valores exclusivos em uma escala contínua, como no histograma de faixas etárias que visualizamos anteriormente, a plotagem do histograma funcionará relativamente bem “de forma imediata”, com argumentos padrão. No entanto, quando o número de valores exclusivos é quase igual ao número de bins, os resultados podem ser duvidosos. O número padrão de bins é 10, enquanto, na característica `PAY_1`, há 11 valores exclusivos. Em casos assim, é melhor definir manualmente o número de bins do histograma para que ele seja igual ao número de valores exclusivos.

Em nosso exemplo atual, já que há poucos valores nos bins mais altos de `PAY_1`, a plotagem pode não ter uma aparência tão diferente. Porém, em geral, é importante lembrar-se disso ao plotar histogramas.

Bordas dos bins

Os locais das bordas dos bins determinam como os valores foram agrupados no histograma. Em vez de indicar o número de bins para a função de plotagem, alternativamente você poderia fornecer uma lista ou um array de números para o argumento de palavra-chave `bins`. Essa entrada seria interpretada como os locais das bordas dos bins no eixo x. É importante entendermos a maneira como os valores são agrupados nos bins no `matplotlib`, com o uso dos locais das bordas. Todos os bins, exceto o último, agrupam valores tão baixos quanto os da borda esquerda e que vão até, **mas sem incluir**, valores tão altos quanto os da borda direita. Em outras palavras, a borda esquerda é fechada, mas a direita é aberta. No entanto, o último bin inclui as duas bordas; ele tem as bordas esquerda e direita fechadas. Isso tem muita importância na prática quando agrupamos uma quantidade relativamente pequena de valores exclusivos que podem coincidir com as bordas dos bins.

Para termos controle sobre a aparência da plotagem, em geral é melhor especificar os locais das bordas dos bins. Criaremos um array de 12 números, que resultará em 11 bins, dedicados a cada um dos valores exclusivos de `PAY_1`:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins
```

A saída mostra os locais das bordas dos bins:

```
pay_1_bins = np.array(range(-2,10)) - 0.5  
pay_1_bins  
array([-2.5, -1.5, -0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  
        8.5])
```

Figura 1.58 – Especificando as bordas dos bins do histograma.

Como última observação sobre o estilo, é importante sempre *rotular suas plotagens* para que elas sejam interpretáveis. Não fizemos isso manualmente, porque, em alguns casos, o `pandas` o faz automaticamente, e, em outros, apenas deixamos as plotagens não rotuladas. De agora em diante, seguiremos a melhor prática e rotularemos todas as plotagens. Usaremos as funções `xlabel` e `ylabel` do `matplotlib` para adicionar rótulos de eixos a essa plotagem. O código é o seguinte:

```
df[pay_feats[0]].hist(bins=pay_1_bins)  
plt.xlabel('PAY_1')  
plt.ylabel('Number of accounts')
```

A saída deve ser esta:

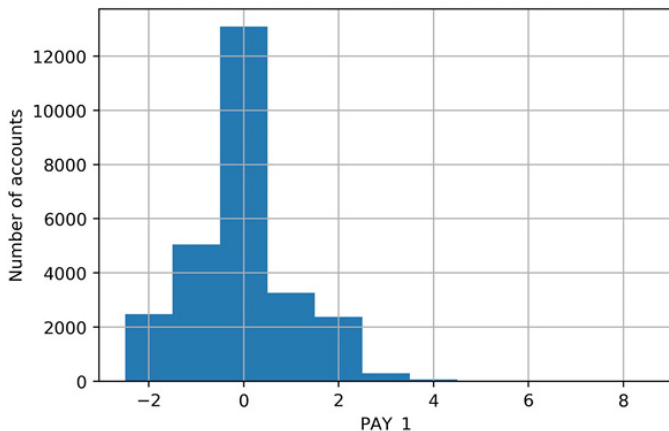


Figura 1.59 – Um histograma melhor para PAY_1.

Embora seja tentador, e com frequência suficiente, chamar as funções de plotagem com os argumentos padrão, uma de suas funções como cientista de dados é criar *visualizações de dados precisas e representativas*. Para isso, às vezes é necessário manipular os detalhes do código de plotagem, como fizemos aqui.

O que aprendemos com essa visualização de dados?

Como vimos nas contagens de valores, essa visualização confirma que a maioria das contas está em boa situação (valores -2, -1 e 0). Para as que não estão, é mais comum que o “atraso em meses” tenha um número menor. Isso faz sentido; provavelmente, a maioria das pessoas está pagando sua dívida sem demorar muito. Caso contrário, sua conta pode ser fechada ou passada para uma agência de cobrança. Examinar a distribuição das características e verificar se ela parece razoável é algo bom para se confirmar com o cliente, já que a qualidade desses dados formará a base da modelagem preditiva que você pretende executar.

Agora que estabelecemos um bom estilo de plotagem para os histogramas, usaremos o pandas para plotar vários histogramas em conjunto e visualizar as características de status de pagamento para cada um dos últimos seis meses. Podemos passar nossa lista de nomes de colunas `pay_feats` para acessar várias colunas a serem plotadas com o método `.hist()`, especificando as bordas de bins que já determinamos e indicando que gostaríamos de uma grade de plotagens 2 por 3. Primeiro, definiremos um tamanho de fonte suficientemente pequeno para caber entre essas **subplotagens**. Aqui está o código que fará isso:

```
mpl.rcParams['font.size'] = 4
```

```
df[pay_feats].hist(bins=pay_1_bins, layout=(2,3))
```

Os títulos das plotagens foram criados automaticamente para nós com base nos nomes das colunas. Os eixos y são considerados como contagens. As visualizações resultantes são as seguintes:

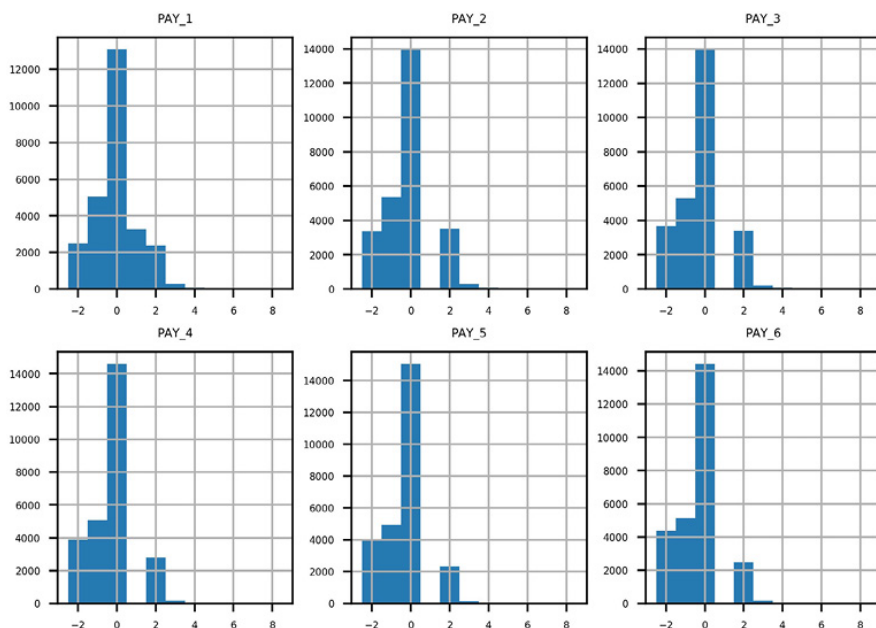


Figura 1.60 – Grade de subplotsagens do histograma.

Já vimos a primeira dessas visualizações, e ela faz sentido. E o resto? Lembre-se das definições dos valores inteiros positivos dessas características e do que cada característica significa. Por exemplo, PAY_2 é o status de reembolso em agosto, PAY_3 é o status de reembolso em julho, e as outras vão voltando no tempo. Um valor igual a 1 significa atraso de 1 mês no pagamento, enquanto o valor 2 significa atraso de dois meses e assim por diante.

Notou que algo não parece certo? Considere os valores entre julho (PAY_3) e agosto (PAY_2). Em julho, poucas contas tiveram atraso de 1 mês no pagamento; essa barra não pode ser vista no histograma. No entanto, em agosto, repentinamente há milhares de contas com atraso de 2 meses no pagamento. Isso não faz sentido: o número de contas com atraso de 2 meses em um mês específico deveria ser menor ou igual ao número de contas com atraso de 1 mês no mês anterior. Examinaremos com mais detalhes as contas com atraso de 2 meses em agosto e veremos qual foi o status de pagamento em julho. Podemos fazer isso com o código a seguir, usando uma máscara booleana e .loc,

como mostrado neste fragmento:

```
df.loc[df['PAY_2'] == 2, ['PAY_2', 'PAY_3']].head()
```

A saída desse código é:

```
df.loc[df['PAY_2'] == 2, ['PAY_2', 'PAY_3']].head()
```

	PAY_2	PAY_3
0	2	-1
1	2	0
13	2	2
15	2	0
47	2	2

Figura 1.61 – Status de pagamento em julho (PAY_3) de contas com atraso de 2 meses no pagamento em agosto (PAY_2).

Na *Figura 1.61*, fica claro que as contas com atraso de 2 meses em agosto têm valores absurdos para o status de pagamento de julho. A única maneira de chegarmos a um atraso de 2 meses seria haver um atraso de um mês no mês anterior, mas nenhuma dessas contas indica isso.

Quando você vir algo assim nos dados, terá de verificar a lógica da consulta usada para criar o dataset ou entrar em contato com a pessoa que o forneceu. Após confirmar os resultados, por exemplo usando `.value_counts()` para visualizar os números diretamente, entramos em contato com nosso cliente para perguntar sobre esse problema.

O cliente informou que está tendo problemas para obter dados do último mês, o que tem gerado relatórios incorretos para contas que têm atraso de 1 mês no pagamento. Em setembro, ele resolveu grande parte dos problemas (mas não totalmente; é por isso que há valores faltando na característica `PAY_1`, como descobrimos). Logo, em nosso dataset, o valor 1 foi subnotificado em todos os meses exceto setembro (a característica `PAY_1`). Teoricamente, o cliente poderia criar uma consulta para fazer uma nova pesquisa em seu banco de dados e determinar os valores corretos para `PAY_2`, `PAY_3` e assim por diante até `PAY_6`. No entanto, por razões práticas, ele não poderá concluir essa análise retrospectiva a tempo de a recebermos e incluímos em nossa análise.

Portanto, só o mês mais recente de nossos dados de status de pagamento está correto. Isso significa que, de todas as características de status de pagamento, só `PAY_1` é representativa de dados futuros, aqueles que serão usados para fazermos previsões com o modelo que

desenvolvemos. Esse é um ponto-chave: *um modelo preditivo depende da obtenção do mesmo tipo de dado para fazer as previsões para as quais foi treinado*. Ou seja, podemos usar PAY_1 como característica em nosso modelo, mas não PAY_2 ou as outras características de status de pagamento de meses anteriores.

Esse episódio mostra a importância de uma verificação abrangente da qualidade dos dados. Só descobrimos o problema vasculhando cuidadosamente os dados. Seria bom se o cliente tivesse informado antecipadamente que estava tendo problemas de relatório nos últimos meses, quando nosso dataset foi coletado, e que o procedimento de geração de relatórios não era **consistente** durante esse período. No entanto, no fim das contas é nossa responsabilidade construir um modelo confiável, logo, temos de nos certificar se os dados estão corretos, fazendo esse tipo de exame cuidadoso. Explicaremos ao cliente que não podemos usar as características mais antigas, já que elas não são representativas dos dados futuros nos quais o modelo **se baseará** (isto é, fará previsões sobre meses futuros) e solicitaremos educadamente que nos informem sobre qualquer problema adicional nos dados do qual tenham conhecimento. No momento não há nenhum.

Atividade 1: Explorando as características financeiras restantes do dataset

Nessa atividade, você examinará as características financeiras restantes de maneira semelhante a como examinamos PAY_1, PAY_2, PAY_3 e assim por diante. Para visualizar melhor alguns desses dados, usaremos uma função matemática que deve ser familiar: o logaritmo. Você usará o pandas com apply, que serve para aplicar qualquer função a uma coluna ou DataFrame inteiro. Ao concluir a atividade, deverá ter o conjunto a seguir de histogramas de transformações logarítmicas de pagamentos diferentes de zero:

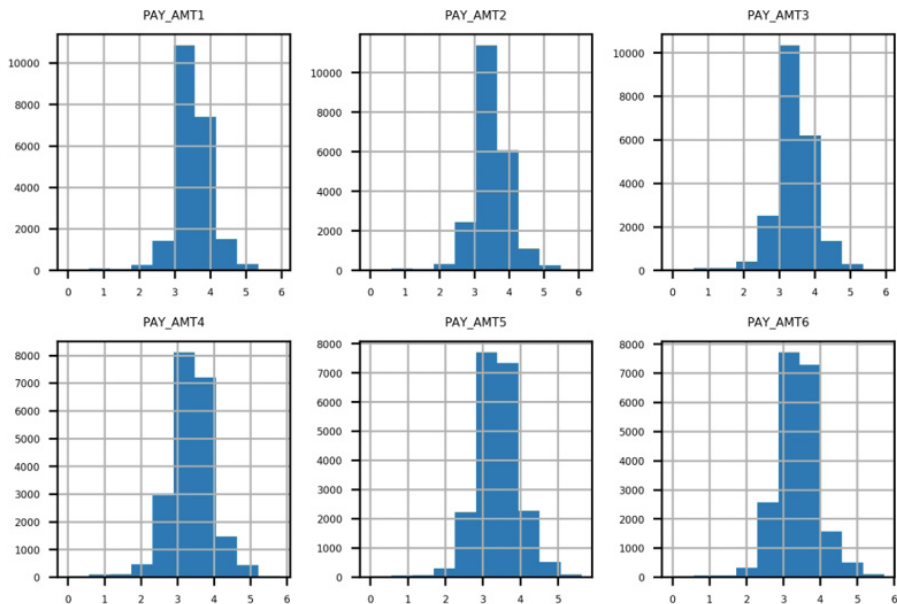


Figura 1.62 – Conjunto esperado de histogramas.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e o gráfico de saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2TXZmrA>.

1. Crie listas com nomes para as características financeiras restantes.
2. Use `.describe()` para examinar as sínteses estatísticas das características de valor da fatura. Reflita sobre o que viu. Faz sentido?
3. Visualize as características de valor da fatura usando uma grade 2 por 3 de plotagens de histograma.
Dica: Você pode usar 20 bins para essa visualização.
4. Obtenha o resumo de `.describe()` para as características de valor do pagamento. Faz sentido?
5. Plote um histograma das características de pagamento da fatura semelhante ao das características de valor da fatura, mas aplique também alguma rotação aos rótulos do eixo x com o argumento de palavra-chave `xrot` para que eles não se sobreponham. Podemos incluir o argumento de palavra-chave `xrot=<ângulo>` em qualquer função de plotagem para girar os rótulos do eixo x de acordo com um ângulo específico em graus. Considere os

resultados.

6. Use uma máscara booleana para ver quantos dos dados de valor do pagamento são exatamente iguais a 0. O resultado faz sentido dado o histograma da etapa anterior?
7. Ignorando os pagamentos iguais a 0 usando a máscara que criou na etapa anterior, utilize o método `.apply()` do pandas e o método `np.log10()` do NumPy para plotar histogramas de transformações logarítmicas dos pagamentos diferentes de zero. Considere os resultados.

Dica: Você pode usar `.apply()` para aplicar qualquer função, inclusive `log10`, a todos os elementos de um DataFrame ou de uma coluna usando a sintaxe a seguir: `.apply(<nome_função>)`.

Nota: A solução dessa atividade pode ser encontrada na página 290.

Resumo

Este foi o primeiro capítulo de nosso livro, *Projetos de ciência de dados com Python*. Aqui, usamos bastante o pandas para carregar e explorar os dados do estudo de caso. Aprendemos como verificar a consistência básica e a precisão usando uma combinação de sínteses estatísticas e visualizações. Respondemos a perguntas como “Os IDs de conta exclusivos são realmente exclusivos?”, “Há dados ausentes que receberam um valor de preenchimento?” e “Os valores das características fazem sentido dada sua definição?”.

Observe que passamos quase o capítulo inteiro identificando e corrigindo problemas em nosso dataset. Com frequência, esse é o estágio mais demorado de um projeto de ciência de dados. Embora nem sempre seja a parte mais empolgante do trabalho, ela fornece os materiais brutos necessários para a construção de modelos e insights interessantes. Esses serão os assuntos de grande parte do resto do livro.

O domínio de ferramentas de software e conceitos matemáticos é o que nos permite executar projetos de ciência de dados, em um nível técnico. No entanto, gerenciar o relacionamento com os clientes, que contam com nossos serviços para gerar insights a partir dos dados, é igualmente importante para um projeto bem-sucedido. Você deve usar o máximo possível o conhecimento que o sócio da empresa tem dos

dados. Provavelmente ele estará mais familiarizado, a não ser que você seja um especialista nos dados do projeto que está executando. Contudo, mesmo nesse caso, sua primeira etapa deve ser uma revisão abrangente e crítica dos dados que está usando.

Em nossa exploração dos dados, descobrimos um problema que poderia ter prejudicado o projeto: os dados que recebemos não eram consistentes internamente. A maioria dos meses das características de status de pagamento estava contaminada por um problema de relatório de dados, incluía valores absurdos e não era representativa dos dados mensais mais recentes ou dos dados que estariam disponíveis para o modelo avançar. Só descobrimos esse problema examinando cuidadosamente todas as características. Embora nem sempre isso seja possível em vários projetos, principalmente quando há um número muito grande de características, você deve tentar inspecionar quantas características puder. Se não puder examinar todas, será útil verificar algumas de cada categoria (se as características se encaixarem em categorias, como financeiras ou demográficas).

Ao discutir problemas de dados como esse com seu cliente, certifique-se de ser respeitoso e profissional. O cliente pode simplesmente ter esquecido o problema ao apresentar os dados. Ou talvez soubesse sobre ele, mas presumiu por alguma razão que não afetaria a análise. Seja como for, você está prestando um serviço essencial mostrando o problema e explicando por que seria incorreto usar dados inadequados para construir um modelo. Você deve apoiar suas alegações em resultados se possível, mostrando que usar dados incorretos leva a um desempenho fraco, ou inalterado, do modelo. Ou, alternativamente, poderia explicar que se um tipo de dado diferente só estiver disponível no futuro, em comparação com os que estão disponíveis no momento para o treinamento, o modelo construído agora não será útil. Seja tão específico quanto puder, apresentando os tipos de gráficos e tabelas que usamos aqui para descobrir o problema nos dados.

No próximo capítulo, examinaremos a variável de resposta de nosso problema de estudo de caso, o que conclui a exploração inicial dos dados. Em seguida, começaremos a ganhar experiência prática com modelos de machine learning e aprenderemos como decidir se um modelo é ou não útil.

Introdução ao Scikit-Learn e avaliação do modelo

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Explicar a variável de resposta
- Descrever as implicações de dados desbalanceados na classificação binária
- Dividir os dados em conjuntos de treinamento e teste
- Descrever o ajuste do modelo no scikit-learn
- Derivar várias métricas para a classificação binária
- Criar uma curva ROC e uma curva precision-recall

Este capítulo concluirá a análise exploratória inicial e apresentará novas ferramentas para execução da avaliação do modelo.

Introdução

O primeiro capítulo o preparou para executar algumas operações básicas com Python e o equipou com ferramentas para a exploração de dados. Especificamente, executamos operações como carregar o dataset e verificar a integridade dos dados; também executamos a primeira análise exploratória no dataset de nosso estudo de caso.

Neste capítulo, concluiremos nossa exploração dos dados examinando a variável de resposta. Após concluirmos que os dados têm alta qualidade e fazem sentido, estaremos prontos para passar para as preocupações práticas do desenvolvimento de modelos de machine learning. Daremos nossos primeiros passos com o scikit-learn, um dos mais populares pacotes de machine learning disponíveis na linguagem

Python. Antes de aprender os detalhes de como os modelos matemáticos funcionam no próximo capítulo, aqui começaremos a nos familiarizar com a sintaxe para usá-los no scikit-learn.

Também aprenderemos algumas técnicas comuns para responder à pergunta “Esse modelo é ou não é bom?”. Há muitas maneiras de executar a avaliação de um modelo. Para aplicações empresariais, algum tipo de análise financeira para determinar o valor que poderia ser criado pelo modelo geralmente é necessário. No entanto, deixaremos isso para o fim do livro.

Há vários critérios importantes para a avaliação de modelos que são considerados conhecimento padrão em ciência de dados e machine learning. Abordaremos aqui algumas das mais usadas métricas de desempenho de modelos de classificação, para lhe dar uma base sólida.

Examinando a variável de resposta e concluindo a exploração inicial

Já investigamos todas as **características** não apenas ver se algum dado está faltando, mas também para verificá-las de um modo geral. Elas são importantes por serem as **entradas** de nosso algoritmo de machine learning. No outro lado do modelo fica a **saída**, uma previsão da **variável de resposta**. Para nosso problema, ela é uma flag binária indicando se uma conta ficará ou não inadimplente no próximo mês, que seria outubro no histórico do dataset.

Nossa missão maior nesse projeto é criar um modelo preditivo para esse objetivo. Já que a variável de resposta é uma flag sim/não, esse problema chama-se tarefa de **classificação binária**. Em nossos dados rotulados, consideramos que as amostras (contas) que ficarão inadimplentes (isto é, com status 'default payment next month' [falta de pagamento no próximo mês] = 1) pertencem à **classe positiva**, enquanto as que não ficarão pertencem à **classe negativa**. A informação mais importante que devemos examinar com relação à resposta de um problema de classificação binária é: qual é a proporção da classe positiva? É uma verificação fácil.

Antes de fazer essa verificação, é essencial carregarmos os pacotes requeridos que usaremos neste capítulo. Isso pode ser feito com o código a seguir:

```
import numpy as np #cálculo numérico
import pandas as pd #preparação dos dados
import matplotlib.pyplot as plt #pacote de plotagem
#A próxima linha ajuda a renderizar plotagem
%matplotlib inline
import matplotlib as mpl #adiciona a funcionalidade de plotagem
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
```

Também precisamos da versão limpa dos dados do estudo de caso. Você pode carregá-la usando este código:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Nota: O dataset limpo deve ter sido salvo como resultado de seu trabalho no *Capítulo 1, Exploração e limpeza de dados*. O caminho dos dados limpos incluído no fragmento de código anterior pode ser diferente dependendo de seu ambiente.

Para encontrar a proporção da classe positiva, precisamos apenas obter a média da variável de resposta no dataset inteiro. Ela terá a interpretação da taxa de inadimplência. Também é útil verificar o número de amostras em cada classe. Isso é apresentado no screenshot a seguir:

```
df['default payment next month'].mean()

0.2217971797179718

df.groupby('default payment next month')['ID'].count()

default payment next month
0      20750
1       5914
Name: ID, dtype: int64
```

Figura 2.1 – Balanceamento de classes da variável de resposta.

Já que a variável alvo é 1 ou 0, a obtenção da média dessa coluna indica o percentual de contas que ficaram inadimplentes: 22%. Também confirmamos o número de contas de cada classe executando uma operação groupby/count. A proporção de amostras na classe positiva (inadimplência = 1), também chamada de **fração da classe**, é uma estatística importante. Na classificação binária, os datasets são descritos considerando-se se eles estão **balanceados** ou **desbalanceados**: as proporções de amostras positivas e negativas são ou não iguais? A maioria dos modelos de classificação de machine learning é projetada para operar com dados balanceados: uma divisão 50/50 entre as classes.

No entanto, na prática, raramente os dados reais estão balanceados. Consequentemente, há vários métodos que têm como objetivo lidar

com dados desbalanceados. Eles são os seguintes:

- **Subamostragem (undersampling)** da classe majoritária: eliminar aleatoriamente amostras da classe majoritária até as frações das classes serem iguais, ou no mínimo menos desbalanceadas.
- **Sobreamostragem (oversampling)** da classe minoritária: adicionar aleatoriamente amostras duplicadas da classe minoritária para atingir o mesmo objetivo.
- **Ponderação de amostras (weighting samples)**: Esse método é executado como parte da etapa de treinamento, para que a classe minoritária receba coletivamente a mesma “ênfase” da classe majoritária no modelo ajustado. O efeito é semelhante ao do oversampling.

Também existem outros métodos mais sofisticados.

Embora nossos dados não estejam balanceados, é bom ressaltar que uma fração de classe positiva de 22% não é particularmente um desbalanceamento. Normalmente, algumas áreas, como a de detecção de fraudes, lidam com frações de classes positivas muito menores: da ordem de 1% ou menos. Isso ocorre porque a proporção de “malfeitores” é muito pequena em comparação com o conjunto total de transações; ao mesmo tempo, é importante conseguir identificá-los se possível. Para problemas como esse, usar um método para resolver o desbalanceamento de classes leva a resultados substancialmente melhores.

Exploramos a variável de resposta, logo, concluímos nossa exploração inicial dos dados. No entanto, a exploração dos dados deve ser considerada uma tarefa contínua que você deve ter sempre em mente durante qualquer projeto. À medida que criamos modelos e geramos novos resultados, é recomendável pensar no que esses resultados informam sobre os dados, o que em geral requer uma volta rápida à fase de exploração. Um tipo particularmente útil de exploração, que normalmente também é feita antes da construção do modelo, é examinar o relacionamento entre as características e a resposta. Fornecemos uma amostra disso no *Capítulo 1, Exploração e limpeza de dados*, quando fizemos o agrupamento pela característica EDUCATION e examinamos a média da variável de resposta. Usaremos esse recurso de novo posteriormente. Contudo, ele está mais ligado à construção de um modelo do que à verificação da qualidade inerente dos dados.

A leitura inicial de todos os dados, como fizemos aqui, é uma tarefa importante a ser executada no começo de um projeto. Ao executá-la,

você deve fazer as seguintes perguntas:

- Os dados estão **completos**?

Há valores ausentes ou outras anomalias?

- Os dados são **consistentes**?

A distribuição muda com o tempo, e, se muda, isso é esperado?

- Os dados **fazem sentido**?

Os valores das características correspondem à sua definição no dicionário de dados?

As duas últimas perguntas o ajudarão a determinar se você acha que os dados estão **corretos**. Se a resposta para uma dessas perguntas for “não,” isso deve ser corrigido antes de o projeto continuar.

Se você também se lembrar de alguma alternativa, ou dado adicional que possa ser útil, e seja possível obtê-lo, agora seria um bom momento no ciclo de vida do projeto para aumentar o dataset. Alguns exemplos seriam dados demográficos de nível de código postal, que você poderia **associar** ao seu dataset, se tivesse os endereços associados às contas. Não temos isso nos dados do estudo de caso e decidimos prosseguir com esse projeto com os dados que já temos.

Introdução ao scikit-learn

Embora o pandas poupe muito tempo no carregamento, exame e limpeza de dados, os algoritmos de machine learning que nos permitirão executar a modelagem preditiva estão localizados em outros pacotes. Consideramos o scikit-learn como o principal pacote de machine learning para Python, com exceção dos de deep learning. Mesmo sendo impossível para qualquer pacote oferecer “tudo”, o scikit-learn chega perto ao acomodar uma ampla variedade de abordagens para classificação e regressão, e aprendizado não supervisionado. Dito isso, outros pacotes que você também deveria conhecer são:

- **SciPy:**

- A maioria dos pacotes que usamos até agora na verdade faz parte do ecossistema SciPy.
- O SciPy também oferece funções leves para abordagens clássicas como regressão linear e programação linear.

- **StatsModels:**

- Orientado para a estatística e mais fácil para usuários familiarizados com o R
- Consegue obter valores-p e intervalos de confiança em coeficientes de regressão
- Suporta modelos de séries temporais como o ARIMA

- **XGBoost:**

- Oferece um modelo de combinações (ensemble model) de última geração, o qual com frequência supera as florestas aleatórias

- **TensorFlow, Keras e PyTorch:**

- Recursos de deep learning

Há muitos outros pacotes Python que podem ser úteis, mas esses servem como exemplo. Passemos então para o scikit-learn, que oferece vários modelos diferentes para regressão e classificação (assim como para o aprendizado não supervisionado), mas, convenientemente, a sintaxe usada é consistente. Nesta seção, ilustraremos a sintaxe usando um modelo de **regressão logística**. Apesar de seu nome, na verdade a regressão logística é um modelo de classificação. Trata-se de um dos mais simples (e, portanto, mais importantes) modelos de classificação. Examinaremos os detalhes matemáticos de como a regressão logística funciona posteriormente. Até lá, você pode considerá-la apenas como uma caixa preta que consegue aprender a partir de dados rotulados para, então, fazer previsões.

Anteriormente, nos familiarizamos com o conceito de treinar um algoritmo a partir de dados para usar o modelo treinado a fim de fazer previsões com novos dados. O scikit-learn encapsula essas funcionalidades básicas no método `.fit` para o treinamento de modelos e no método `.predict` para a execução de previsões. Devido à sintaxe consistente, você pode chamar `.fit` e `.predict` em qualquer modelo do scikit-learn, seja de regressão linear ou de árvores de classificação.

A primeira etapa é selecionar algum modelo, nesse exemplo uma regressão logística, e instanciá-lo a partir da **classe** fornecida pelo scikit-learn. Isso significa que você estará obtendo o blueprint do modelo que o scikit-learn disponibiliza e criando um **objeto** útil a partir dele. Você pode treinar esse objeto com seus dados e salvá-lo em disco para uso posterior. Os fragmentos de código a seguir podem ser usados para a execução dessa tarefa. Primeiro temos de importar a

classe:

```
from sklearn.linear_model import LogisticRegression
```

O código que instancia a classe para criar um objeto é:

```
my_lr = LogisticRegression()
```

Agora o objeto é uma variável em nosso espaço de trabalho. Podemos examiná-la usando o código a seguir:

```
my_lr
```

Ele deve fornecer esta saída:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, max_iter=100, multi_class='warn',
                    n_jobs=None, penalty='l2', random_state=None, solver='warn',
                    tol=0.0001, verbose=0, warm_start=False)
```

Figura 2.2 – Instanciando um modelo no scikit-learn.

Observe que, após criarmos o objeto de modelo, o examinamos e vimos uma extensa saída. Essa saída representa as **opções padrão** que vêm junto com o modelo de regressão logística. Na verdade, são escolhas que fizemos relacionadas aos detalhes de implementação do modelo, sem tomar conhecimento disso. O perigo de um pacote fácil de usar como o scikit-learn é que ele oculta essas escolhas; pudemos instanciar o modelo sem especificar nenhuma delas. No entanto, sempre que você usar um modelo de machine learning preparado de antemão como os modelos do scikit-learn, a primeira coisa a fazer é conhecer todas as opções que estão disponíveis. Uma prática recomendada nesses casos é fornecer explicitamente cada parâmetro de palavra-chave para o modelo ao criar o objeto. Mesmo se selecionar todas as opções padrão, isso o ajudará a ter uma noção melhor das decisões que está tomando.

Aqui está o código para instancição de um modelo de regressão logística com todas as opções padrão:

```
my_new_lr = LogisticRegression(C=1.0, class_weight=None, dual=False,
                                fit_intercept=True,
                                intercept_scaling=1, max_iter=100, multi_class='warn',
                                n_jobs=None, penalty='l2', random_state=None, solver='warn',
                                tol=0.0001, verbose=0, warm_start=False)
```

Ainda que o objeto que criamos aqui em `my_new_lr` seja idêntico a `my_lr`, agir dessa maneira explícita será particularmente útil quando você estiver começando a travar contato com diferentes tipos de modelos. Quando estiver mais familiarizado, poderá executar a instancição com as opções padrão e fazer alterações posteriormente

conforme necessário. Mostraremos como isso pode ser feito. O código a seguir define duas opções e exibe o estado atual do objeto de modelo:

```
my_new_lr.C = 0.1
my_new_lr.solver = 'liblinear'
my_new_lr
```

Ele deve produzir o seguinte:

```
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

Figura 2.3 – Atualizando o hiperparâmetro e o solver do modelo.

Aqui, pegamos o que é conhecido como **hiperparâmetro** do modelo, C, e o atualizamos de seu valor padrão 1 para 0.1. Também especificamos um solver. Você deve estar curioso para saber o que é um hiperparâmetro, o que é C e o que todas as outras opções significam. Isso é bom; é preciso saber essas coisas. É suficiente dizer que um hiperparâmetro é como uma opção que você fornece para o modelo, antes de ajustá-lo aos dados. Posteriormente, explicaremos com detalhes o que são essas opções e como você pode selecionar valores para elas.

Por enquanto, apenas para ilustrar a funcionalidade básica, avançaremos sem maiores referências e ajustaremos essa regressão logística quase padrão a alguns dados. Como você já sabe, os algoritmos de aprendizado supervisionado dependem de dados rotulados. Isso significa que precisamos tanto das características, normalmente contidas em uma variável chamada X, quanto das respostas correspondentes, em uma variável chamada y. Tomaremos emprestadas de nosso dataset as 10 primeiras amostras de uma única característica, e a resposta, para ilustrar:

```
X = df['EDUCATION'][0:10].values.reshape(-1,1)
X
```

Esse código deve exibir os valores da característica EDUCATION para as 10 primeiras amostras:

```
array([[2],
       [2],
       [2],
       [2],
       [2],
       [1],
       [1],
       [2],
       [3],
       [3]])
```

Figura 2.4 – Dez primeiros valores de uma característica.

Os 10 primeiros valores correspondentes da variável de resposta podem ser obtidos assim:

```
y = df['default payment next month'][0:10].values
y
```

```
array([1, 1, 0, 0, 0, 0, 0, 0, 0, 0])
```

Figura 2.5 – Dez primeiros valores da variável de resposta.

Aqui, selecionamos duas séries (isto é, colunas) de nosso DataFrame: a característica EDUCATION que estamos discutindo, e a variável de resposta. Em seguida, selecionamos os 10 primeiros elementos de cada uma e, para concluir, usamos o método `.values` para retornar arrays NumPy. Observe também que usamos o método `.reshape` para redimensionar as características. O scikit-learn espera que a primeira dimensão (isto é, o número de linhas) do array de características seja o número de amostras, logo, tivemos de fazer esse redimensionamento para X, mas não para y. O `-1` no primeiro argumento posicional de `.reshape` torna a forma do array de saída flexível nessa dimensão, de acordo com o número de dados que entrarem. Já que temos apenas uma característica nesse exemplo, especificamos o número de colunas como o segundo argumento, `1`, e deixamos que o argumento `-1` indique que o array deve “ser preenchido” ao longo da primeira dimensão com quantos elementos forem necessários para acomodar os dados, nesse caso 10 elementos. Observe que, embora tenhamos extraído os dados para inseri-los em arrays NumPy e mostrar como isso pode ser feito, também poderíamos ter usado séries do pandas como entrada para o scikit-learn.

Agora usaremos esses dados para ajustar nossa regressão logística. Isso pode ser feito com uma única linha:

```
my_new_lr.fit(X, y)
```

```
LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
intercept_scaling=1, max_iter=100, multi_class='warn',
n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
tol=0.0001, verbose=0, warm_start=False)
```

Figura 2.6 – Ajustando um modelo no scikit-learn.

É tudo o que precisamos saber sobre isso! Quando seus dados estiverem preparados, ajustar o modelo será somente consequência. É claro que por enquanto estamos ignorando todas as opções importantes e o que elas significam. No entanto, tecnicamente falando, é muito fácil ajustar um modelo no que se refere ao código. É possível ver que a saída dessa célula apenas exibe as opções novamente. Embora não tenhamos retornado nada do procedimento de ajuste a não ser essa saída, uma alteração muito importante ocorreu. Agora o objeto de modelo `my_new_lr` é um modelo treinado. Podemos dizer que essa alteração ocorreu **in loco**, já que não foi criado um novo objeto; o objeto existente, `my_new_lr`, foi modificado. Isso é semelhante a modificar um `DataFrame` *in loco*. Já podemos usar nosso modelo treinado para fazer previsões para novas características que o modelo ainda não “viu”. Tentaremos usar as próximas 10 linhas da característica `EDUCATION`.

Podemos selecionar e visualizar essas características usando uma nova variável, `new_X`:

```
new_X = df['EDUCATION'][10:20].values.reshape(-1,1)
new_X
```

```
array([[3],
       [1],
       [2],
       [2],
       [1],
       [3],
       [1],
       [1],
       [1],
       [3]])
```

Figura 2.7 – Novas características para as quais serão feitas previsões.

As previsões são feitas assim:

```
my_new_lr.predict(new_X)
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Figura 2.8 – Previsões para as novas características.

Também podemos visualizar quais são os valores reais correspondentes a essas previsões:

```
df['default payment next month'][10:20].values
```

```
array([0, 0, 0, 1, 0, 0, 1, 0, 0, 0])
```

Figura 2.9 – Os rótulos reais das previsões.

Mostramos várias coisas aqui. Após obter os valores de nossas novas características, chamamos o método `.predict` no modelo treinado. Observe que o único argumento desse método é um conjunto de características, isto é, um “X” que chamamos de `new_X`. Nesse caso, não foi fornecido um “y”. Na verdade, é para isso que serve a modelagem preditiva: você não sabe necessariamente o valor verdadeiro da variável de resposta, logo, tem de poder prevê-lo. Vimos que a saída do método `.predict` são as previsões do modelo para essas amostras. Em seguida, comparamos essas previsões com os valores reais da variável de resposta, que extraímos do `DataFrame`.

Então, como se saiu nosso modelo construído às cegas? Podemos observar superficialmente que, como o modelo previu todos os 0s, e 80% dos rótulos verdadeiros eram 0s, acertamos em 80% das vezes, o que parece muito bom. Por outro lado, não conseguimos prever com sucesso nenhum 1. Logo, se eles fossem importantes, não teríamos nos saído tão bem. Embora esse seja apenas um pequeno exemplo para familiarizá-lo com como o `scikit-learn` funciona, é útil considerar o que seria uma “boa” previsão para esse problema. Veremos os detalhes de avaliação da capacidade do modelo preditivo em breve. Por enquanto, parabéns por ter praticado com dados reais e ajustado seu primeiro modelo de machine learning.

Gerando dados sintéticos

Agora você já está familiarizado com vários pacotes Python. Na verdade, eles são suficientes para a execução de um amplo conjunto de projetos de ciência de dados, da classificação à regressão e ao aprendizado não supervisionado. E também viu como funciona o ajuste de um modelo no `scikit-learn`, um dos pacotes de machine learning mais úteis em Python. No próximo exercício, percorrerá por conta própria o processo de ajuste de um modelo.

Para ter dados para ajustar, você gerará seus próprios **dados sintéticos**, uma ferramenta de aprendizado valiosa para ilustrar conceitos matemáticos. Para criar dados sintéticos, mostraremos novamente como usar a biblioteca `random` do `NumPy` para a geração de números aleatórios, assim como as funções `scatter` e `plot` do `matplotlib` para criar as plotagens de dispersão e linear, respectivamente. No exercício, usaremos o `scikit-learn` para a parte de

regressão linear. Não veremos os detalhes da regressão linear aqui, embora certamente você já deva estar familiarizado com o conceito de uma **linha de melhor ajuste**, que é a mesma coisa.

Para começar, usaremos o NumPy para criar um array unidimensional de valores de características chamado *X* e composto de 1.000 números reais aleatórios (em outras palavras, não só inteiros como também decimais) entre 0 e 10. Empregaremos novamente um **seed** para o gerador de números aleatórios. Em seguida, usaremos `numpy.random.uniform`, que se baseia na distribuição uniforme: é igualmente provável que selecione qualquer número entre *low* (inclusive) e *high* (sem incluí-lo); ele retornará um array com o tamanho (*size*) que você especificar. Criaremos um array unidimensional (isto é, um vetor) com 1.000 elementos e depois examinaremos os 10 primeiros. Tudo isso pode ser feito com o uso do código a seguir:

```
np.random.seed(seed = 1)
X = np.random.uniform(low=0.0, high=10.0, size=(1000,))
X[0:10]
```

A saída deve ser a seguinte:

```
array([4.17022005e+00, 7.20324493e+00, 1.14374817e-03, 3.02332573e+00,
       1.46755891e+00, 9.23385948e-01, 1.86260211e+00, 3.45560727e+00,
       3.96767474e+00, 5.38816734e+00])
```

Figura 2.10 – Criando números aleatórios uniformemente distribuídos com o NumPy.

Dados para uma regressão linear

Agora precisamos de uma variável de resposta. Para esse exemplo, geraremos dados que se baseiam nas suposições da regressão linear: obedecem a uma tendência linear, mas tem erros normalmente distribuídos. É improvável que um dataset do mundo real atenda às suposições estatísticas formais de um modelo como esse, mas você pode criar um conjunto de dados sintéticos que o faça. Teoricamente, a regressão linear só deve ser usada para modelar dados em que a resposta seja uma transformação linear das características, com ruído normalmente distribuído (também chamado de gaussiano):

$$y = ax + b + N(\mu, \sigma)$$

Figura 2.11 – Equação linear com ruído gaussiano.

Aqui, *a* é a inclinação, *b* é a interceptação, e o ruído gaussiano tem

média igual a μ com desvio padrão de σ . Para escrever um código que implemente isso, temos de criar um vetor de respostas correspondente, y , calculado como a inclinação vezes o array de características, X , mais algum ruído gaussiano (novamente com o uso do NumPy), e uma interceptação.

Para criar os dados da regressão linear, primeiro definiremos o seed aleatório. Em seguida, declararemos variáveis para a inclinação e a interceptação de nossos dados lineares. Para concluir, criaremos a variável de resposta usando a equação familiar de uma linha, com o acréscimo de algum ruído gaussiano: um array de 1.000 pontos de dados com a mesma dimensão (size) do array de características, X , em que a média do ruído (loc) é 0 e o desvio padrão (scale) é 1. Isso adicionará uma pequena “expansão” aos dados lineares. O código para a criação dos dados lineares com ruído gaussiano é:

```
np.random.seed(seed = 1)
slope = 0.25
intercept = -1.25
y = slope * X + np.random.normal(loc=0.0, scale=1.0, size=(1000,)) + intercept
```

Agora queremos visualizar os dados. Usaremos o matplotlib para plotar y em relação à característica X como uma plotagem dispersa. Primeiro utilizaremos .rcParams para definir a resolução (dpi = dots per inch [pontos por polegada]) para uma imagem de boa nitidez. Em seguida, criaremos a plotagem dispersa com plt.scatter, em que os dados X e y são os dois primeiros argumentos, respectivamente, e o argumento s recebe um tamanho para os pontos.

Esse código pode ser usado para a plotagem:

```
mpl.rcParams['figure.dpi'] = 400
plt.scatter(X,y,s=1)
```

Após executar essas células, você deve ver algo como a Figura 2.12 em seu notebook:

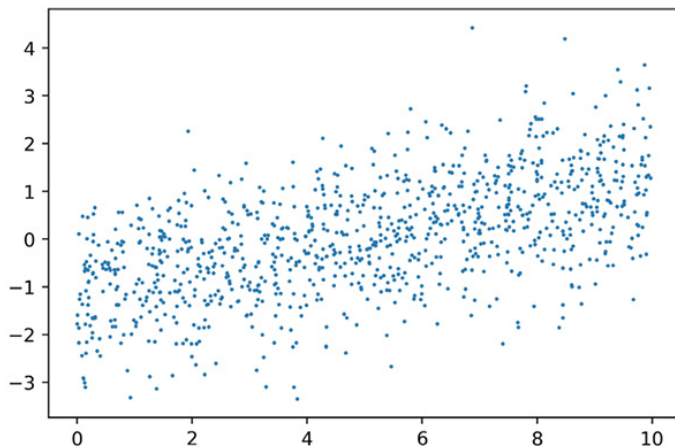


Figura 2.12 – Plote o relacionamento linear ruidoso.

Parecem alguns dados lineares ruidosos, como esperávamos! Agora temos de modelá-los.

Exercício 8: Regressão linear com o scikit-Learn

Nesse exercício, pegaremos os dados sintéticos que acabamos de gerar e determinaremos uma linha de melhor ajuste, ou regressão linear, usando o scikit-learn. A primeira etapa é importar uma classe de modelo de regressão linear do scikit-learn e criar um objeto a partir dela. A importação é semelhante à da classe `LogisticRegression` que importamos anteriormente. Como em qualquer classe de modelo, você deve observar todas as opções padrão. Repare que, nesse caso, para uma regressão linear, não há tantas opções a serem especificadas: você pode usar os padrões para esse exercício. As configurações padrão incluem `fit_intercept=True`, significando que o modelo de regressão terá um termo para a interceptação. Isso é, sem dúvida, apropriado, já que adicionamos uma interceptação aos dados sintéticos. Execute as etapas a seguir para fazer o exercício:

Nota: Para os Exercícios 8–10 e a Atividade 2, o código e a saída resultante foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2UWPgYo>. Você pode rolar até a seção apropriada dentro do Jupyter Notebook para localizar o exercício ou a atividade que quiser.

1. Execute este código para importar a classe de modelo de regressão linear e instanciá-la:

```
from sklearn.linear_model import LinearRegression  
lin_reg = LinearRegression()
```


lin_reg

Você deve ver esta saída:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

Figura 2.13 – Trabalhando com a classe de regressão linear.

Agora podemos ajustar o modelo usando nossos dados sintéticos, lembrando-nos de redimensionar o array de características, como fizemos anteriormente, para que as amostras fiquem ao longo da primeira dimensão. Após ajustar o modelo de regressão linear, examinaremos `lin_reg.intercept_`, que contém a interceptação do modelo ajustado, como era de se esperar, e `lin_reg.coef_`, que contém a inclinação.

2. Execute este código para ajustar o modelo e examinar os coeficientes:

```
lin_reg.fit(X.reshape(-1,1), y)  
print(lin_reg.intercept_)  
print(lin_reg.coef_)
```

Você deve ver esta saída para a interceptação e a inclinação:

```
-1.161254600282509  
[0.24602508]
```

Figura 2.14 – Ajustando a regressão e examinando os parâmetros.

Podemos ver novamente que ajustar um modelo no scikit-learn, após os dados estarem preparados e as opções do modelo terem sido definidas, é um processo trivial. Isso ocorre porque o usuário não precisa fazer todo o trabalho algorítmico de determinar os parâmetros do modelo. Discutiremos alguns desses detalhes posteriormente no caso do modelo de regressão logística que usaremos nos dados do estudo de caso.

E quanto à inclinação e à interceptação de nosso modelo ajustado?

Esses números estão bem próximos da inclinação e da interceptação que indicamos ao criar o modelo. No entanto, devido ao ruído aleatório, são apenas aproximações.

Finalmente podemos usar o modelo para fazer previsões de valores das características. Aqui faremos isso empregando os mesmos dados utilizados no ajuste do modelo: o array de características, `X`. Capturaremos a saída como uma variável chamada `y_pred`: esse caso é muito parecido com o anterior, só que agora estamos

inserindo a saída do método `.predict` em uma variável.

3. Execute este código para fazer previsões:

```
y_pred = lin_reg.predict(X.reshape(-1,1))
```

Podemos exibir as previsões, `y_pred`, da característica `X` como uma plotagem linear sobre a plotagem dispersa dos dados da característica e da resposta como fizemos na *Etapa 4*. Aqui repetimos a *Etapa 4*, com o acréscimo de `plt.plot`, que produz uma plotagem linear por padrão, para plotar os valores da característica e da resposta prevista pelo modelo. Observe que após os dados `X` e `y` inserimos `'r'` em nossa chamada a `plt.plot`. Esse argumento de palavra-chave dá à linha a cor vermelha e faz parte de uma sintaxe abreviada para a formatação da plotagem.

4. Esse código pode ser usado para plotar os dados brutos e as previsões do modelo ajustado feitas para estes dados:

```
plt.scatter(X,y,s=1)  
plt.plot(X,y_pred,'r')
```

Após executar essa célula, você deve ver algo assim:

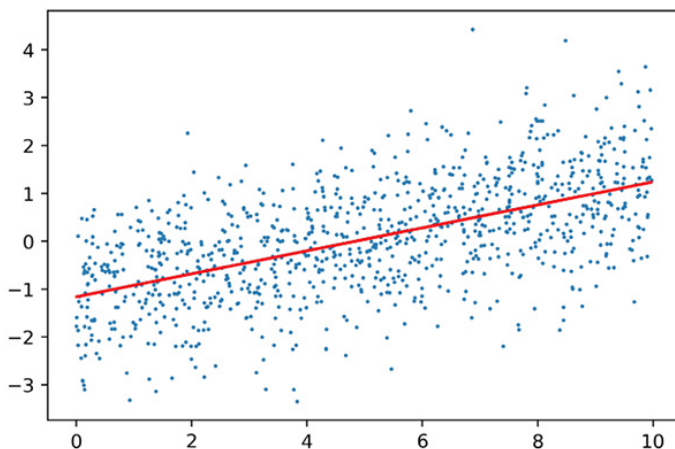


Figura 2.15 – Plotando os dados e a linha de regressão.

A plotagem exibe como seria a linha de melhor ajuste para os dados, como esperávamos.

Nesse exercício, ao contrário de quando chamamos `.predict` com a regressão logística, fizemos previsões para os mesmos dados `X` que usamos para treinar o modelo. Essa é uma diferença importante. Enquanto no caso atual estamos vendo como o modelo “ajusta” os mesmos dados com os quais foi treinado, anteriormente examinamos previsões do modelo para dados novos e desconhecidos. Em machine

learning, geralmente o que nos interessa são os recursos preditivos: queremos modelos que nos ajudem a saber os possíveis resultados de futuros cenários. No entanto, no fim das contas as previsões tanto dos **dados de treinamento** usados para ajustar o modelo, quanto dos **dados de teste**, que não foram usados no ajuste, são importantes para entendermos o funcionamento do modelo. Formalizaremos essas noções posteriormente no *Capítulo 4, O trade-off entre viés e variância* quando discutirmos o **trade-off entre viés e variância**.

Métricas de desempenho de modelos para a classificação binária

Antes de começar realmente a construir modelos preditivos, queremos saber como podemos determinar, após criar um modelo, se ele é de alguma forma “bom”. Como era de se esperar, essa questão tem recebido muita atenção de pesquisadores e profissionais da área. Consequentemente, há uma grande variedade de métricas de desempenho de modelos para escolhermos.

Nota: Para conhecer as diversas opções, acesse a página de avaliação de modelos do scikit-learn: https://scikit-learn.org/stable/modules/model_evaluation.html#model-evaluation.

Ao selecionar uma métrica de desempenho para avaliar a qualidade preditiva de um modelo, é importante nos lembrarmos de duas coisas.

Conveniência da métrica para o problema

Normalmente as métricas são definidas apenas para uma classe específica de problemas, como de classificação ou regressão. Para um problema de classificação binária, várias métricas podem representar a precisão da pergunta de resposta sim ou não que o modelo responde. Um nível adicional de detalhe nesse caso é com que frequência o modelo está correto para cada classe, as classes positiva e negativa. Veremos essas métricas com detalhes aqui. Por outro lado, as métricas de regressão visam medir o quanto uma previsão se aproximou do valor alvo. Se estivermos tentando prever o preço de uma casa, o quão perto chegamos? Estamos sistematicamente superestimando ou subestimando? Estamos avaliando as casas mais caras erroneamente e as mais baratas corretamente? Há muitas maneiras possíveis de se examinarem métricas de regressão.

A métrica responde à pergunta da empresa?

Independentemente da classe de problemas na qual você estiver trabalhando, haverá muitas opções para a métrica. Qual é a correta? E, mesmo assim, como saber se um modelo é “suficientemente bom” em termos de métrica? De certa forma, essa é uma pergunta subjetiva. No entanto, podemos ser objetivos ao considerar qual é a meta do modelo. Em um contexto empresarial, as metas típicas são gerar lucro ou reduzir perdas. Basicamente, você precisa tornar complementares o problema da empresa, que com frequência está relacionado a questões financeiras, e a métrica que usará para julgar seu modelo. Por exemplo, em nosso problema de não pagamento de dívidas, há um custo particularmente alto associado à não identificação correta das contas que ficarão inadimplentes? Isso é mais importante do que identificar corretamente as contas que não estarão inadimplentes?

Posteriormente no livro, incorporaremos o conceito de custos e benefícios relativos de classificações corretas e incorretas em nosso problema e conduziremos uma análise financeira. Primeiro, introduziremos as métricas mais comuns usadas para a avaliação da qualidade preditiva de modelos de classificação binária, os tipos de modelos que temos de construir para nosso estudo de caso.

Dividindo os dados: conjuntos de treinamento e de teste

Na introdução ao scikit-learn deste capítulo, apresentamos o conceito de uso de um modelo treinado para fazermos previsões para novos dados que o modelo ainda não tinha “visto”. Verificamos que esse é um conceito básico em modelagem preditiva. Em nossa tentativa de criar um modelo que tenha capacidade preditiva, precisamos de alguma maneira de medir com que eficiência o modelo pode fazer previsões para dados não usados para ajustá-lo. Isso ocorre porque no ajuste de um modelo, este se torna “especializado” no aprendizado do relacionamento entre as características e a resposta do conjunto específico de dados rotulados que foram usados no ajuste. Embora isso seja bom, queremos poder usar o modelo para fazer previsões precisas para novos dados não vistos, cujo valor real dos rótulos não é conhecido.

Por exemplo, em nosso estudo de caso, quando entregarmos o modelo treinado para o cliente, ele gerará um novo dataset de características como as que temos agora, só que, em vez de se estenderem pelo período de abril a setembro, elas irão de maio a outubro. O cliente

usará o modelo com essas características para prever se as contas ficarão inadimplentes em novembro.

Para saber o nível de eficiência que podemos esperar de nosso modelo na previsão das contas que ficarão inadimplentes em novembro (o que não será conhecido até dezembro), podemos pegar nosso dataset atual e reservar alguns dos dados que temos, com rótulos conhecidos, do processo de treinamento do modelo. Esses dados são chamados de **dados de teste**, mas também podem ser chamados de **dados fora da amostra** já que são compostos de amostras que não foram usadas no treinamento do modelo. As amostras usadas para treinar o modelo são chamadas de **dados de treinamento**. Separar um conjunto de dados de teste nos dará uma ideia de como será o desempenho do modelo quando ele for usado para sua finalidade, fazer previsões para amostras que não foram incluídas durante seu treinamento. Neste capítulo, criaremos um exemplo de conjunto de treinamento/teste para ilustrar diferentes métricas da classificação binária.

Usaremos a conveniente funcionalidade `train_test_split` do `scikit-learn` para dividir os dados de modo que 80% sejam usados para treinamento e 20% sejam reservados para teste. Esses percentuais são uma maneira comum de fazer a divisão; em geral, devemos ter dados de treinamento suficientes para permitir que o algoritmo “aprenda” adequadamente a partir de uma amostra de dados representativa. No entanto, os percentuais não são rígidos. Se você tiver um número muito grande de amostras, talvez não precise de um percentual tão alto de dados de treinamento, já que poderá chegar a um conjunto de treinamento bastante extenso e representativo com um percentual mais baixo. Recomendamos que faça testes com diferentes tamanhos e veja o efeito. Além disso, lembre-se de que cada problema é diferente no que diz respeito a quantos dados são necessários para o treinamento efetivo de um modelo. Não há uma regra rígida para o dimensionamento dos conjuntos de treinamento e teste. Considere o código mostrado no fragmento a seguir:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    df['EDUCATION'].values.reshape(-1,1), df['default payment next month'].values,
    test_size=0.2, random_state=24)
```

Observe no fragmento anterior que definimos `test_size` como 0.2, ou 20%. O tamanho dos dados de treinamento será definido automaticamente com o restante, 80%. Examinaremos as dimensões de nossos dados de treinamento e teste para ver se elas são as

esperadas, como mostrado na saída a seguir:

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

(21331, 1)
(5333, 1)
(21331,)
(5333,)
```

Figura 2.16 – Dimensão dos conjuntos de treinamento e teste.

Você deve confirmar por conta própria se o número de amostras (linhas) dos conjuntos de treinamento e teste é consistente com a divisão 80/20.

Ao fazer a divisão treinamento/teste, também definimos o parâmetro `random_state`, que é um seed de número aleatório. O uso desse parâmetro permite uma divisão treinamento/teste consistente entre as execuções do notebook. Caso contrário, o procedimento de divisão aleatória selecionaria um percentual de 20% dos dados de teste diferente a cada vez que o código fosse executado. Para concluir, observe que, como em nossa ilustração inicial do ajuste do modelo, usaremos uma única característica aqui: EDUCATION. Estamos fazendo isso apenas para mostrar como dividir dados e calcular diferentes métricas de desempenho do modelo. Usaremos outras características posteriormente quando examinarmos os detalhes de como funcionam os diferentes modelos de machine learning e tentarmos criar o melhor modelo que pudermos para entregar para nosso cliente.

O primeiro argumento de `train_test_split` são as características, nesse caso apenas EDUCATION, e o segundo é a resposta. Há quatro saídas: as características das amostras dos conjuntos de treinamento e teste, respectivamente, e as variáveis de resposta correspondentes a esses conjuntos de características. O que essa função fez foi simplesmente selecionar aleatoriamente 20% dos índices brutos do dataset e criar um subconjunto dessas características e respostas como dados de teste, deixando o resto como treinamento. Agora que temos nossos dados de treinamento e teste, temos de nos certificar se sua natureza é a mesma entre esses conjuntos. Especificamente, temos de saber se a fração da classe positiva é semelhante. Você pode ver isso na saída a seguir:

```
np.mean(y_train)
```

```
0.223102526838873
```

```
np.mean(y_test)
```

```
0.21657603600225014
```

Figura 2.17 – Frações das classes nos dados de treinamento e teste.

As frações da classe positiva tanto nos dados de treinamento quanto nos de teste são de aproximadamente 22%. Isso é bom, já que são iguais às dos dados gerais e podemos dizer que o conjunto de treinamento é representativo do conjunto de teste. Nesse caso, como temos um dataset muito grande com dezenas de milhares de amostras, e as classes não estão tão desbalanceadas, não precisamos tomar precauções para assegurar que isso ocorra.

No entanto, podemos deduzir que, se o dataset for menor, e a classe positiva muito rara, pode ocorrer de as frações das classes serem perceptivelmente diferentes entre os conjuntos de treinamento e teste, ou, ainda pior, não haver amostras positivas no conjunto de teste! Para evitar esses cenários, você poderia usar uma **amostragem estratificada**, com o argumento de palavra-chave `stratify` de `train_test_split`. Esse procedimento também faz uma divisão aleatória dos dados em conjuntos de treinamento e teste, mas garante que as frações das classes sejam iguais ou muito semelhantes entre os dois.

Nota: Teste out-of-time – Se seus dados contiverem tanto características quanto respostas estendendo-se por um período de tempo significativo, é recomendável tentar fazer a divisão treinamento/teste com o passar do tempo. Por exemplo, se você tiver dois anos de dados com características e respostas para cada mês, talvez seja melhor treinar o modelo sequencialmente com 12 meses de dados e testar com o mês seguinte, ou com o mês depois dele, dependendo do que for operacionalmente viável quando o modelo for usado. Você poderia repetir isso até exaurir seus dados, para obter resultados de teste diferentes. Essa abordagem lhe daria insights úteis sobre o desempenho do modelo, porque simula as condições reais que ele encontrará quando for implantado: um modelo treinado com características e respostas mais antigas será usado para fazer previsões de dados mais novos. No estudo de caso, as respostas vêm de um único ponto no tempo (as inadimplências dentro de um mês), logo, essa não é uma opção.

Acurácia da classificação

Agora daremos prosseguimento e ajustaremos um exemplo de modelo para ilustrar métricas de classificação binária. Continuaremos a usar uma regressão logística com opções quase padrão, selecionando as mesmas opções que demonstramos no *Capítulo 1, Exploração e limpeza de dados*:

```
from sklearn.linear_model import LogisticRegression

example_lr = LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                                intercept_scaling=1, max_iter=100, multi_class='warn',
                                n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
                                tol=0.0001, verbose=0, warm_start=False)
```

Figura 2.18 – Carregando a classe do modelo e criando um objeto de modelo.

Treinaremos então o modelo, como era de se esperar, usando os dados rotulados de nosso conjunto de treinamento. Em seguida, usaremos imediatamente o modelo treinado para fazer previsões para as características das amostras do conjunto reservado para teste:

```
example_lr.fit(X_train, y_train)

LogisticRegression(C=0.1, class_weight=None, dual=False, fit_intercept=True,
                    intercept_scaling=1, max_iter=100, multi_class='warn',
                    n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
                    tol=0.0001, verbose=0, warm_start=False)

y_pred = example_lr.predict(X_test)
```

Figura 2.19 – Treinando um modelo e fazendo previsões no conjunto de teste.

Armazenamos os rótulos do conjunto de teste previstos pelo modelo em uma variável chamada `y_pred`. Como devemos avaliar agora a qualidade dessas previsões? Temos os rótulos reais, na variável `y_test`. Primeiro, calcularemos a que provavelmente é a mais simples de todas as métricas de classificação binária: a **acurácia**. A **acurácia** é definida como a proporção de amostras que foram classificadas corretamente.

Uma maneira de calcular a acurácia é criar uma máscara lógica que seja igual a `True` sempre que o rótulo previsto for igual ao rótulo real; caso contrário, ela será igual a `False`. Em seguida, podemos calcular a média dessa máscara, que interpretará `True` como 1 e `False` como 0, fornecendo a proporção de classificações corretas:


```
is_correct = y_pred == y_test

np.mean(is_correct)

0.7834239639977498
```

Figura 2.20 – Calculando a acurácia da classificação com uma máscara lógica.

Isso indica que o modelo está correto 78% das vezes. Embora esse seja um cálculo muito simples, na verdade há maneiras mais fáceis de calcular a acurácia com o conveniente uso do scikit-learn. Uma maneira é usar o método `.score` do modelo treinado, passando as características dos dados de teste para fazer previsões e os rótulos de teste. Esse método faz as previsões e, em seguida, executa o mesmo cálculo que usamos anteriormente. Ou poderíamos importar a biblioteca `metrics` do scikit-learn, que contém muitas métricas de desempenho de modelos, inclusive `accuracy_score`. Para fazê-lo, é preciso passar os rótulos reais e os previstos:

```
example_lr.score(X_test, y_test)

0.7834239639977498

from sklearn import metrics

metrics.accuracy_score(y_test, y_pred)

0.7834239639977498
```

Figura 2.21 – Calculando a acurácia da classificação com o scikit-learn.

Essas opções fornecem o mesmo resultado, como esperado. Agora que sabemos o nível de precisão do modelo, como interpretar essa métrica? À primeira vista, uma acurácia de 78% pode parecer boa. A maioria das previsões está correta. No entanto, um teste importante da acurácia na classificação binária é a comparação dos dados com um modelo hipotético muito simples que faz uma única previsão: esse modelo prevê a classe majoritária de cada amostra, independentemente de quais sejam as características. Embora na prática seja um modelo inútil, ele fornece um caso extremo importante com o qual podemos comparar a acurácia de nosso modelo treinado. Às vezes, esses casos extremos são chamados de modelos nulos.

Considere qual seria a acurácia desse modelo nulo. Em nosso dataset, sabemos que aproximadamente 22% das amostras são positivas. Logo, a classe negativa é a majoritária, com os 78% das amostras restantes. Portanto, um modelo nulo para esse dataset, que sempre prevê a classe majoritária negativa, estará certo 78% das vezes. Ao

compararmos nosso modelo treinado com esse modelo nulo, fica claro que na verdade uma acurácia de 78% não é muito útil. Podemos obter a mesma acurácia com um modelo que não se preocupe com as características!

Embora possamos interpretar a acurácia em relação a um modelo nulo de classe majoritária, há outras métricas de classificação binária que se aprofundam um pouco mais em como o modelo está se saindo para amostras negativas versus positivas.

Taxa de verdadeiros positivos, taxa de falsos positivos e matriz de confusão

Na classificação binária, apenas dois rótulos são considerados: positivo e negativo. Como uma maneira mais descritiva do que a acurácia da previsão ao longo de todas as amostras para examinarmos o desempenho do modelo, também podemos examinar a acurácia somente das amostras que tenham um rótulo positivo. A proporção das amostras previstas como positivas é chamada de **taxa de verdadeiros positivos** (TPR, **true positive rate**). Se dissermos que **P** é o número de amostras da **classe positiva** nos dados de teste, e **TP** é o número de **verdadeiros positivos**, definidos como o número de amostras que foram previstas como positivas pelo modelo, então, a TPR será:

$$TPR = \frac{TP}{P}$$

Figura 2.22 – Equação da taxa de verdadeiros positivos.

O inverso da taxa de verdadeiros positivos é a **taxa de falsos negativos** (FNR, **false negative rate**). Trata-se da proporção de amostras de teste positivas que previmos incorretamente como negativas. Esses erros são chamados de **falsos negativos** (FN) e a taxa de falsos negativos (FNR) é calculada da seguinte forma:

$$FNR = \frac{FN}{P}$$

Figura 2.23 – Equação da taxa de falsos negativos.

Já que todas as amostras positivas são previstas de maneira correta ou incorreta, a soma do número de verdadeiros positivos e do número de falsos negativos é igual ao número total de amostras positivas. Matematicamente, $P = TP + FN$ e, portanto, se usarmos as definições de TPR e FNR, teremos o seguinte:

$$TPR + FNR = 1$$

Figura 2.24 – Relação entre a TPR e a FNR.

Como a soma da TPR e da FNR é igual a 1, geralmente é suficiente calcular apenas uma delas.

Da mesma forma que temos a TPR e a FNR, há a **taxa de verdadeiros negativos** (TNR, **true negative rate**) e a **taxa de falsos positivos** (FPR, **false positive rate**). Se N for o número de amostras **negativas**, a soma das amostras **verdadeiras negativas** (TN, **true negative**) será o número dessas amostras que foram previstas corretamente, e a soma das amostras **falsas positivas** (FP) será o número previsto incorretamente como positivo:

$$TNR = \frac{TN}{N}$$

Figura 2.25 – Equação da taxa de verdadeiros negativos.

$$FPR = \frac{FP}{N}$$

Figura 2.26 – Equação da taxa de falsos positivos.

$$TNR + FPR = 1$$

Figura 2.27 – Relação entre a taxa de verdadeiros negativos e a de falsos positivos.

Os verdadeiros e falsos positivos e negativos podem ser convenientemente resumidos em uma tabela chamada **matriz de confusão**. A matriz de confusão de um problema de classificação binária é uma matriz 2 x 2 em que a classe verdadeira fica ao longo de um eixo e a classe prevista fica ao longo do outro eixo. Ela fornece um resumo rápido de quantos verdadeiros e falsos positivos e negativos existem:

		Classe prevista	
		N	P
Classe verdadeira	N	TN	FP
	P	FN	TP

Figura 2.28 – Matriz de confusão da classificação binária.

Já que esperamos fazer classificações corretas, queremos que a **diagonal** (isto é, as entradas ao longo de uma linha diagonal da parte

superior esquerda à parte inferior direita: TN e TP) da matriz de confusão seja relativamente grande, e que as entradas fora da diagonal sejam relativamente menores, porque representam classificações incorretas.

Exercício 9: Calculando as taxas de verdadeiros e falsos positivos e negativos e a matriz de confusão em Python

Neste exercício, usaremos os dados de teste e as previsões do modelo de regressão logística que criamos anteriormente, utilizando apenas a característica EDUCATION. Ilustraremos como calcular manualmente as taxas de verdadeiros e falsos positivos e negativos, assim como os números de verdadeiros e falsos positivos e negativos da matriz de confusão. Em seguida, mostraremos uma maneira rápida de calcular uma matriz de confusão com o scikit-learn. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2UWPgYo>.

1. Execute este código para calcular o número de amostras positivas:

```
P = sum(y_test)
P
```

A saída deve ser esta:

1155

Figura 2.29 – Calculando o número de positivos.

Agora precisamos do número de verdadeiros positivos. São as amostras em que o rótulo verdadeiro é 1 e a previsão também é 1. Podemos identificá-las com uma máscara lógica para as amostras que forem positivas (`y_test == 1`) E (& é o operador lógico **AND** [E, em português] em Python) tiverem uma previsão positiva (`y_pred == 1`).

2. Use este código para calcular o número de verdadeiros positivos:

```
TP = sum( (y_test == 1) & (y_pred == 1) )
TP
```

0

Figura 2.30 – Calculando o número de verdadeiros positivos.

A taxa de verdadeiros positivos é a proporção de verdadeiros

positivos da classe positiva.

3. Execute o código a seguir para obter a taxa de verdadeiros positivos:

```
TPR = TP/P
```

```
TPR
```

Você verá esta saída:

```
0.0
```

Figura 2.31 – Calculando a taxa de verdadeiros positivos.

Da mesma forma, podemos identificar os falsos negativos.

4. Calcule o número de falsos negativos com este código:

```
FN = sum( (y_test == 1) & (y_pred == 0) )
```

```
FN
```

Ele deve exibir o seguinte:

```
1155
```

Figura 2.32 – Calculando o número de falsos negativos.

Também queremos a taxa de falsos negativos.

5. Calcule a taxa de falsos negativos com:

```
FNR = FN/P
```

```
FNR
```

Esse código deve exibir a seguinte saída:

```
1.0
```

Figura 2.33 – Taxa de falsos negativos.

O que descobrimos com as taxas de verdadeiros positivos e falsos negativos?

Primeiro, confirmamos que sua soma é 1. Esse fato é fácil de ver porque a $TPR = 0$ e a $FPR = 1$. O que isso nos diz sobre nosso modelo? No conjunto de teste, pelo menos para as amostras positivas, na verdade o modelo agiu como um modelo nulo de classe majoritária. Todas as amostras positivas foram previstas como negativas, logo, nenhuma delas foi prevista corretamente.

6. Calcularemos a TNR e a FPR de nossos dados de teste. Já que esses cálculos são muito semelhantes aos que examinamos anteriormente, vamos exibi-los de uma só vez e demonstraremos uma nova função Python:

```
N = sum(y_test==0)
N
```

4178

```
TN = sum( (y_test==0) & (y_pred==0))
TN
```

4178

```
FP = sum( (y_test==0) & (y_pred==1))
FP
```

0

```
TNR = TN/N
FPR = FP/N
print('The true negative rate is {} and the false positive rate is {}'.format(TNR, FPR))
```

The true negative rate is 1.0 and the false positive rate is 0.0

Figura 2.34 – Calculando as taxas de verdadeiros negativos e falsos positivos e exibindo-as.

Além de calcular a TNR e a FPR de maneira semelhante a como fizemos com a TPR e a FNR, demonstramos a função print do Python junto com o método .format para strings, que permite a substituição de variáveis em locais marcados com chaves ({}). Há várias opções para a formatação de números, como a inclusão de uma quantidade específica de casas decimais.

Nota: Para ver detalhes adicionais, acesse <https://docs.python.org/3/tutorial/inputoutput.html>.

O que aprendemos aqui? Na verdade, nosso modelo se comporta exatamente como o modelo nulo de classe majoritária para todas as amostras, tanto positivas quanto negativas. É claro que precisaremos de um modelo melhor.

Embora tenhamos calculado manualmente todas as entradas da matriz de confusão nesse exercício, no scikit-learn há uma maneira rápida de fazer isso. Observe que, no scikit-learn, a classe verdadeira fica ao longo do eixo vertical e a classe prevista fica ao longo do eixo horizontal da matriz de confusão, como apresentamos anteriormente.

7. Crie uma matriz de confusão no scikit-learn com este código:

```
metrics.confusion_matrix(y_test, y_pred)
```

Você obterá a saída a seguir:

```
metrics.confusion_matrix(y_test, y_pred)
array([[4178,  0],
       [1155,  0]])
```

Figura 2.35 – Matriz de confusão de nosso exemplo de modelo.

Todas as informações necessárias para o cálculo da TPR, FNR, TNR e FPR estão contidas na matriz de confusão. Repare também que há várias outras métricas de classificação que podem ser derivadas da matriz de confusão. Na verdade, algumas delas são sinônimas das que já examinamos aqui. Por exemplo, a TPR também é chamada de **recall** e **sensibilidade**. Além do recall, outra métrica que é com frequência usada para a classificação binária é a **precisão**: trata-se da proporção de previsões positivas que estão corretas (em vez da proporção de amostras positivas que foram previstas corretamente). Ganharemos mais experiência no uso da precisão na atividade deste capítulo.

Nota: Classificação multiclasse – Nosso estudo de caso envolve um problema de classificação binária com apenas dois resultados possíveis: a conta está ou não inadimplente. Outro tipo importante de problema de classificação do machine learning é a classificação multiclasse. Nessa classificação, há vários resultados mutuamente exclusivos possíveis. Um exemplo clássico é o reconhecimento de dígitos manuscritos; um dígito manuscrito só pode ser um dos dígitos 0, 1, 2, ... 9. Embora a classificação multiclasse não faça parte do escopo deste livro, as métricas que estamos aprendendo agora para a classificação binária podem ser estendidas à configuração multiclasse.

Descobrimos probabilidades previstas: como a regressão logística faz previsões?

Agora que nos familiarizamos com a acurácia, os verdadeiros e falsos positivos e negativos e a matriz de confusão, é hora de aprender um pouco mais sobre a regressão logística, para aumentar nosso conhecimento das métricas de classificação binária. Até aqui, consideramos a regressão logística apenas como uma “caixa preta” que aprende a partir de dados de treinamento rotulados e faz previsões para novas características. Conheceremos o funcionamento da regressão logística com detalhes posteriormente no livro, mas já podemos começar a olhar dentro da caixa preta.

Uma coisa que devemos considerar sobre como a regressão logística funciona é que as previsões brutas – em outras palavras, as saídas diretas da equação matemática que define a regressão logística – não são rótulos binários. Na verdade são **probabilidades** em uma escala de 0 a 1 (embora, tecnicamente, a equação nunca permita que as probabilidades sejam exatamente iguais a 0 ou 1, como veremos). Essas probabilidades só são transformadas em previsões binárias com

o uso de um **limite (threshold)**. O limite é a probabilidade acima da qual uma previsão é declarada como positiva e abaixo da qual ela é negativa. No scikit-learn ele é igual a 0.5. Isso significa que qualquer amostra com probabilidade prevista de pelo menos 0.5 é identificada como positiva e com probabilidade prevista < 0.5 é considerada negativa. No entanto, não somos obrigados a usar o limite padrão. A seleção do limite é uma das principais flexibilidades da regressão logística, assim como em outros algoritmos de classificação de machine learning que estimam probabilidades de associação de classe.

Exercício 10: Obtendo probabilidades previstas a partir de um modelo de regressão logística

No exercício a seguir, nos familiarizaremos com as probabilidades previstas da regressão logística e veremos como obtê-las a partir de um modelo do scikit-learn.

Podemos começar a descobrir probabilidades previstas examinando melhor os métodos disponíveis para nós no objeto de modelo de regressão logística que treinamos neste capítulo. Lembre-se de que antes, uma vez treinado o modelo, podíamos fazer previsões binárias usando os valores das características de novas amostras, passando esses valores para o método `.predict` do modelo treinado. Essas são previsões feitas para a suposição de um limite igual a 0.5.

No entanto, podemos acessar diretamente as probabilidades previstas dessas amostras, usando o método `.predict_proba`. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante desse exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2UWPgYo>.

1. Obtenha as probabilidades previstas para as amostras de teste usando este código:

```
y_pred_proba = example_lr.predict_proba(X_test)
y_pred_proba
```

A saída deve ser:

```
array([[0.77423402, 0.22576598],
       [0.77423402, 0.22576598],
       [0.78792915, 0.21207085],
       ...,
       [0.78792915, 0.21207085],
       [0.78792915, 0.21207085],
       [0.78792915, 0.21207085]])
```


Figura 2.36 – Probabilidades previstas para os dados de teste.

Podemos ver na saída desse código, armazenada em `y_pred_proba`, que há duas colunas. Isso ocorre porque há duas classes em nosso problema de classificação: negativa e positiva. Supondo que os rótulos negativos sejam codificados como 0 e os positivos como 1, como o são em nossos dados, o `scikit-learn` considerará a probabilidade de associação de classe negativa como a primeira coluna e a de associação de classe positiva como a segunda coluna.

Pela nossa discussão das probabilidades, podemos deduzir que a soma das probabilidades das duas classes é igual a 1 para cada amostra. Confirmaremos isso.

Primeiro, podemos usar `np.sum` na primeira dimensão (colunas) para calcular a soma das probabilidades de cada amostra.

2. Calcule a soma das probabilidades previstas para cada amostra com este código:

```
prob_sum = np.sum(y_pred_proba,1)
prob_sum
```

A saída é a seguinte:

```
array([1., 1., 1., ..., 1., 1., 1.])
```

Figura 2.37 – Calculando as somas das probabilidades previstas.

Certamente parece que temos apenas 1s! Devemos verificar se o resultado tem a mesma forma do array de rótulos de dados de teste.

3. Verifique a forma do array com este código:

```
prob_sum.shape
```

Ele deve exibir o seguinte:

```
(5333,)
```

Figura 2.38 – Forma do array de soma das probabilidades.

Ótimo; essa é a forma esperada. Agora verificaremos se todos os valores são iguais a 1. Usaremos `np.unique` para exibir os elementos exclusivos do array. Esse método é semelhante à palavra-chave `DISTINCT` do SQL. Se todas as somas de probabilidades forem realmente iguais a 1, deve haver apenas 1 elemento exclusivo no array de probabilidades: 1.

4. Exiba todos os elementos exclusivos do array com este código:

```
np.unique(prob_sum)
```

A saída a seguir deve ser exibida:

```
array([1.])
```

Figura 2.39 – Valor exclusivo do array de somas das probabilidades.

Após confirmar nossa dedução sobre as probabilidades previstas, já que a soma das probabilidades das classes é 1, é suficiente considerar apenas a segunda coluna, a probabilidade prevista da associação de classe positiva. Vamos capturá-la em um array.

5. Execute este código para inserir a segunda coluna do array de probabilidades previstas (probabilidade prevista de associação de classe positiva) em um array:

```
pos_proba = y_pred_proba[:,1]
pos_proba
```

A saída deve ser esta:

```
array([0.22576598, 0.22576598, 0.21207085, ..., 0.21207085, 0.21207085,
       0.21207085])
```

Figura 2.40 – Probabilidades previstas de associação de classe positiva.

Qual seria a aparência dessas probabilidades? Uma maneira de descobrir, e um bom diagnóstico para a saída do modelo, é plotar as probabilidades previstas. Um histograma seria uma maneira natural de fazermos isso. Podemos usar diretamente a função de histogramas do matplotlib, `hist()`, para criá-lo. Observe que, se você executar uma célula que tenha apenas a função de histograma, a saída da função do NumPy será retornada antes da plotagem. Isso inclui o número de amostras de cada bin e os locais das bordas dos bins.

6. Execute este código para ver a saída da função de histograma e uma plotagem não formatada (não mostrada):

```
plt.hist(pos_proba)
```

A saída é:

```
(array([1883.,    0.,    0., 2519.,    0.,    0.,  849.,    0.,    0.,
        82.]),
 array([0.21207085, 0.21636321, 0.22065556, 0.22494792, 0.22924027,
        0.23353263, 0.23782498, 0.24211734, 0.24640969, 0.25070205,
        0.2549944 ]),
 <a list of 10 Patch objects>)
```

Figura 2.41 – Detalhes do cálculo do histograma.

Essas informações podem ser úteis e também poderiam ser obtidas diretamente a partir da função `np.histogram()`. No entanto, aqui estamos mais interessados na plotagem, logo, ajustaremos o tamanho da fonte e adicionaremos alguns rótulos de eixos.

7. Execute este código para ver a plotagem formatada do histograma das probabilidades previstas:

```
mpl.rcParams['font.size'] = 12
plt.hist(pos_proba)
plt.xlabel('Predicted probability of positive class for testing data')
plt.ylabel('Number of samples')
```

A plotagem deve ficar assim:

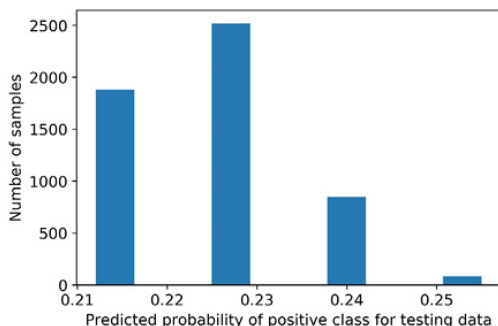


Figura 2.42 – Plotagem do histograma das probabilidades previstas.

Observe que há apenas quatro bins com amostras no histograma de probabilidades e eles estão bem distantes uns dos outros. Isso ocorre porque há somente quatro valores exclusivos para a característica EDUCATION, que é a única característica de nosso exemplo de modelo.

Observe também que todas as probabilidades previstas estão abaixo de 0.5. É por isso que cada amostra foi prevista como negativa, pelo uso do limite 0.5. Podemos deduzir que, se definíssemos nosso limite abaixo de 0.5, obteríamos resultados diferentes. Por exemplo, se definíssemos o limite como 0.25, todas as amostras do bin menor da extrema direita da *Figura 2.42* seriam classificadas como positivas, já que a probabilidade prevista para todas elas estaria acima de 0.25. Seria informativo para nós se pudéssemos visualizar quantas dessas amostras tinham realmente rótulos positivos. Em seguida, poderíamos ver se mover nosso limite para baixo até 0.25 melhoraria o desempenho de nosso classificador pela classificação das amostras do bin da extrema direita como positivas.

Na verdade, podemos visualizar isso facilmente, usando um **histograma empilhado**. Ele será muito parecido com o histograma da *Figura 2.42*, exceto pelo fato de as amostras negativas e positivas terem cores diferentes. Em primeiro lugar, precisamos distinguir as amostras positivas das negativas nas probabilidades previstas.

Podemos fazer isso indexando nosso array de probabilidades previstas com máscaras lógicas; primeiro obteremos as amostras positivas, em que $y_{\text{test}} == 1$ e, em seguida, obteremos as negativas, em que $y_{\text{test}} == 0$.

8. Isole as probabilidades previstas das amostras positivas e negativas com este código:

```
pos_sample_pos_proba = pos_proba[y_test == 1]
neg_sample_pos_proba = pos_proba[y_test == 0]
```

Agora queremos plotar essas probabilidades como um histograma empilhado. O código é semelhante ao do histograma que criamos anteriormente, exceto por passarmos uma lista dos arrays a serem plotados, que são os arrays de probabilidades de amostras positivas e negativas que acabamos de criar, e uma palavra-chave indicando que queremos que as barras sejam empilhadas em vez de plotadas lado a lado. Também criaremos uma legenda para que as cores sejam claramente identificáveis na plotagem.

9. Plote um histograma empilhado com o código a seguir:

```
plt.hist([pos_sample_pos_proba, neg_sample_pos_proba], histtype='barstacked')
plt.legend(['Positive samples', 'Negative samples'])
plt.xlabel('Predicted probability of positive class')
plt.ylabel('Number of samples')
```

A plotagem deve ficar assim:

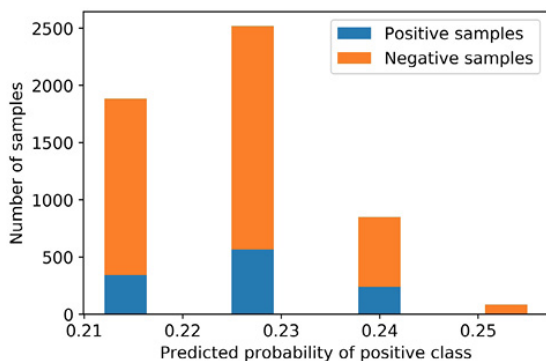


Figura 2.43 – Histograma empilhado de probabilidades previstas por classe.

A plotagem mostra o rótulo verdadeiro das amostras para cada probabilidade prevista. Agora podemos considerar qual seria o efeito de baixarmos o limite para 0.25. Pare um pouco para pensar no que isso significaria, lembrando-se de que qualquer amostra com probabilidade prevista igual ou acima do limite seria classificada

como positiva.

Já que quase todas as amostras do bin menor da direita na *Figura 2.42* são negativas, se diminuíssemos o limite para 0.25, elas seriam classificadas erroneamente como amostras positivas e aumentaríamos nossa taxa de falsos positivos. Ao mesmo tempo, ainda não teríamos conseguido classificar muitas amostras positivas corretamente, caso conseguíssemos classificar alguma, logo, a taxa de verdadeiros positivos aumentaria demais. Parece que fazer essa alteração diminui a acurácia do modelo.

Curva receiver operating characteristic (ROC)

A definição de um limite para o classificador depende de encontrarmos um “ponto de equilíbrio” no qual possamos recuperar com sucesso verdadeiros positivos suficientes sem gerar excesso de falsos positivos. À medida que baixarmos o limite, haverá mais de ambos. Um bom classificador consegue capturar mais verdadeiros positivos sem gerar um número maior de falsos positivos. Qual seria o efeito de baixarmos ainda mais o limite com as probabilidades previstas do exercício anterior? Há um método clássico de visualização em machine learning, com a métrica correspondente, que ajuda a responder a esse tipo de pergunta.

A curva **receiver operating characteristic (ROC, característica de operação do receptor)** é uma plotagem dos pares de taxas de verdadeiros positivos (*eixo y*) e falsos positivos (*eixo x*) que resulta da diminuição do limite de 1 até 0. Podemos deduzir que, se o limite for 1, não haverá previsões positivas, já que a regressão logística só prevê probabilidades estritamente entre 0 e 1 (as extremidades não são incluídas). Como não há previsões positivas, tanto a TPR quanto a FPR são iguais a 0, logo, a curva ROC começa em (0, 0). À medida que o limite for diminuído, a TPR começará a crescer, esperamos que mais rapidamente do que a FPR se tivermos um bom classificador. Quando o limite chegar a 0, todas as amostras serão previstas como positivas, incluindo as que forem realmente positivas, mas também as que forem negativas. Isso significa que a TPR será 1, mas a FPR também será 1. Entre esses dois extremos estarão as opções apropriadas para onde você deve definir o limite, dependendo dos custos e benefícios relativos dos verdadeiros e falsos positivos e negativos para o problema específico que estiver sendo considerado. Dessa forma, é possível obter um cenário completo do desempenho do classificador em todos os diferentes limites para decidirmos qual usar.

Poderíamos escrever o código para determinar as TPRs e FPRs da curva ROC usando as probabilidades previstas e variando o limite de 1 a 0. Em vez disso, usaremos a conveniente funcionalidade do scikit-learn, que receberá os rótulos verdadeiros e as probabilidades previstas como entradas e retornará arrays de TPRs, FPRs e dos limites que levarão a elas. Em seguida, plotaremos as TPRs em relação às FPRs para exibir a curva ROC. Execute esse código para usar o scikit-learn a fim de gerar os arrays de TPRs e FPRs da curva ROC.

```
fpr, tpr, thresholds = metrics.roc_curve(y_test, pos_proba)
```

Agora precisamos produzir uma plotagem. Usaremos `plt.plot`, que criará uma plotagem linear utilizando o primeiro argumento como os valores de x (FPRs), o segundo argumento como os valores de y (TPRs) e a abreviação `'*-'` para indicar uma plotagem linear com símbolos de estrela onde os pontos de dados estiverem localizados. Adicionaremos uma plotagem linear do ponto (0, 0) a (1, 1), que aparecerá em vermelho (`'r'`) e como uma linha tracejada (`'--'`). Também forneceremos uma legenda (que explicaremos em breve) para a plotagem, assim como rótulos para os eixos e um título. Esse código produz a plotagem da curva ROC:

```
plt.plot(fpr, tpr, '*-')
plt.plot([0, 1], [0, 1], 'r--')
plt.legend(['Logistic regression', 'Random chance'])
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.title('ROC curve')
```

E a plotagem deve ter esta aparência:

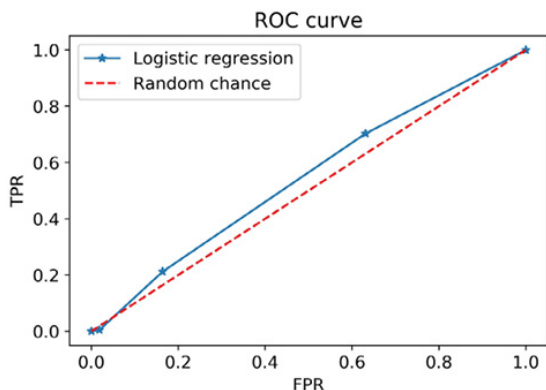


Figura 2.44 – Curva ROC de nossa regressão logística, com uma linha de chance aleatória sendo exibida a título de comparação.

O que aprendemos com nossa curva ROC? Podemos ver que ela

começa em (0,0) com um limite suficientemente alto para que não haja classificações positivas. Em seguida, a primeira coisa que acontece, como imaginamos anteriormente ao diminuir o limite para aproximadamente 0.25, é um aumento na taxa de falsos positivos, mas um aumento muito menor na taxa de verdadeiros positivos. Os efeitos de continuarmos a diminuir o limite para que as outras barras de nosso histograma empilhado sejam incluídas como classificações positivas são mostrados pelos pontos subsequentes na linha. Podemos ver os limites que levam a essas taxas examinando o array de limites, que não faz parte da plotagem. Visualize os limites usados para calcular a curva ROC usando este código:

```
thresholds
```

A saída deve ser esta:

```
thresholds
array([1.2549944 , 0.2549944 , 0.24007604, 0.22576598, 0.21207085])
```

Figura 2.45 – Limites da curva ROC.

Observe que na verdade o primeiro limite está acima de 1; na prática, ele só precisa ser suficientemente alto para que não haja classificações positivas.

Agora considere qual seria a aparência de uma curva ROC “desejável”. À medida que aumentarmos o limite, queremos ver a TPR aumentar, indicando que nosso classificador está fazendo um bom trabalho de identificar corretamente amostras positivas. Ao mesmo tempo, idealmente a FPR não deve aumentar tanto. A curva ROC de um classificador eficaz “estreitaria” o canto superior esquerdo da plotagem: alta taxa de verdadeiros positivos, baixa taxa de falsos positivos. Podemos deduzir que um classificador perfeito obteria uma TPR igual a 1 (recuperação de todas as amostras positivas) e uma FPR igual a 0, e apareceria como um tipo de quadrado começando em (0,0), indo até (0,1) e terminando em (1,1). Embora na prática esse tipo de desempenho seja altamente improvável, ele nos mostra um caso limite.

Considere também qual seria a área sob a curva (AUC, area under the curve) desse classificador, lembrando-se das integrais do cálculo integral se as obteve. A AUC de um classificador perfeito seria 1, porque a forma da curva seria um quadrado sobre o intervalo de unidades [0, 1].

Por outro lado, a linha chamada “Random chance” em nossa plotagem

é a curva ROC que teoricamente resultaria do arremesso de uma moeda não viciada como classificador: tem a mesma probabilidade de fornecer um verdadeiro positivo e um falso positivo, logo, a diminuição do limite introduz mais de cada um em igual proporção e a TPR e a FPR aumentam na mesma taxa. A AUC sob essa ROC teria a metade da do classificador perfeito, como você pode ver graficamente, e seria igual a 0.5.

Logo, em geral, a AUC da curva ROC fica entre 0.5 e 1 (embora valores abaixo de 0.5 sejam tecnicamente possíveis). Valores próximos a 0.5 indicam que, como classificador, o modelo não pode fazer mais do que calcular uma chance aleatória (arremesso da moeda), enquanto valores mais próximos de 1 indicam um desempenho melhor. A **ROC AUC** é uma métrica-chave para a avaliação da qualidade de um classificador e é amplamente usada em machine learning. Ela também é chamada de **estatística-C**.

Já que é uma métrica tão importante, é claro que o scikit-learn tem uma maneira conveniente de calculá-la. Veremos qual é a ROC AUC do classificador de regressão logística passando as mesmas informações que fornecemos para a função `roc_curve`. Calcule a área sob a curva ROC com este código:

```
metrics.roc_auc_score(y_test, pos_proba)
```

E observe a saída:

```
0.5434630477972642
```

Figura 2.46 – ROC AUC da regressão logística.

A ROC AUC da regressão logística chega perto de 0.5, o que significa que ela não é um classificador muito eficaz. Isso não é surpresa, considerando-se que não tivemos trabalho para determinar que características do pool de candidatas são realmente úteis nesse momento. Estamos apenas nos acostumando à sintaxe de ajuste de modelos e aprendendo a maneira de calcular métricas de qualidade usando um modelo simples contendo apenas a característica EDUCATION. Posteriormente, ao considerar outras características, esperamos obter uma ROC AUC mais alta.

Nota: Curva ROC: por que ela recebeu esse nome? – Durante a Segunda Guerra Mundial, operadores do receptor (ro, receiver operators) de sinais de radar eram avaliados por sua habilidade em julgar se algo que apareceu na tela era ou não uma aeronave inimiga. Essas decisões envolviam os mesmos conceitos de verdadeiros

e falsos positivos e negativos nos quais estamos interessados na classificação binária. A curva ROC foi criada como uma maneira de se medir a eficácia dos operadores de equipamentos de recepção de sinais de radar.

Precisão

Antes de passar para a atividade, consideraremos a métrica de classificação introduzida sem detalhes anteriormente: a **precisão**. Como a curva ROC, esse diagnóstico é útil para um intervalo de limites. A precisão é definida da seguinte forma:

$$precisão = \frac{TP}{TP + FP}$$

Figura 2.47 – Equação da precisão.

Considere a interpretação dessa equação, no que diz respeito a variarmos o limite ao longo do conjunto de probabilidades previstas, como fizemos para a curva ROC. Em um limite alto, haverá relativamente poucas amostras previstas como positivas. À medida que diminuirmos o limite, cada vez mais amostras serão previstas como positivas. O esperado é que, ao fazermos isso, o número de verdadeiros positivos aumente mais rapidamente do que o número de falsos positivos, como vimos na curva ROC. A precisão examina a proporção de verdadeiros positivos em relação à soma de verdadeiros e falsos positivos. Pensemos no denominador: o que é a soma de verdadeiros e falsos positivos?

Na verdade, essa soma é o número total de previsões positivas. Logo, a precisão mede a proporção de previsões positivas que estão corretas entre todas as previsões positivas. Portanto, também é chamada de **valor preditivo positivo**. Geralmente a precisão é usada quando as classes estão muito desbalanceadas e se preocupa mais com a obtenção de um bom desempenho do modelo referente à exatidão das previsões positivas. É fácil obter uma alta taxa de verdadeiros positivos quando prevemos muitas amostras como positivas, mas isso também aumenta o número de falsos positivos relativos ao número de previsões positivas. Se houver muito poucas amostras positivas, a precisão fornecerá uma avaliação mais crítica da qualidade de um classificador do que a ROC AUC. Como ocorre com a curva ROC, há uma função conveniente no scikit-learn para calcularmos a precisão, junto com o recall (também conhecido como taxa de verdadeiros positivos), para um intervalo de limites: `metrics.precision_recall_curve`.

Por que a precisão pode ser uma medida útil de desempenho do classificador? Suponhamos que, para cada previsão positiva do modelo, você tomasse algum curso de ação mais caro, como oferecer um cupom de desconto para um cliente a fim de fidelizá-lo. Os falsos positivos seriam os clientes para os quais você ofereceu um cupom, mas não precisava fazê-lo, representando dinheiro perdido. Você vai querer se certificar de tomar as decisões certas referentes a para quem deve oferecer cupons. A precisão seria uma boa métrica para ser usada nessa situação.

Atividade 2: Executando a regressão logística com uma nova característica e criando uma curva precision-recall

Nesta atividade, você treinará uma regressão logística usando uma característica diferente de EDUCATION. Em seguida, avaliará graficamente o tradeoff entre precisão e recall, assim como calculará a área sob uma curva precision-recall. Você também calculará a ROC AUC nos conjuntos tanto de treinamento quanto de teste e os comparará.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e a saída resultante dessa atividade foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2UWPgYo>.

1. Use o método `train_test_split` do scikit-learn para criar um novo conjunto de dados de treinamento e de teste. Dessa vez, em vez de EDUCATION, use LIMIT_BAL: o limite de crédito da conta.
2. Treine um modelo de regressão logística usando os dados de treinamento provenientes da divisão.
3. Crie o array de probabilidades previstas para os dados de teste.
4. Calcule a ROC AUC usando as probabilidades previstas e os rótulos verdadeiros dos dados de teste. Compare o resultado com a ROC AUC da característica EDUCATION.
5. Plote a curva ROC.
6. Calcule os dados da **curva precision-recall** baseando-se nos dados de teste e usando a funcionalidade do scikit-learn.
7. Plote a curva precision-recall usando o matplotlib.
8. Use o scikit-learn para calcular a área sob a curva precision-recall.

9. Agora recalcule a ROC AUC, mas, dessa vez, faça-o para os dados de treinamento. Em que esse cálculo é diferente, conceitual e quantitativamente, do que fizemos antes?

Nota: A solução para essa atividade pode ser encontrada na página 296.

Resumo

Neste capítulo, terminamos a exploração inicial dos dados do estudo de caso examinando a variável de resposta. Após nos certificarmos de que o dataset estava completo e correto, nos sentimos preparados para explorar a relação entre características e resposta e construir modelos.

Passamos grande parte do capítulo nos acostumando com o ajuste de modelos no scikit-learn em um nível técnico, de codificação, e conhecendo as métricas que poderíamos usar com o problema de classificação binária do estudo de caso. Ao fazer testes com diferentes conjuntos de características e tipos de modelos, você precisará de alguma maneira de saber se uma abordagem está funcionando melhor do que a outra. Consequentemente, terá de usar métricas de desempenho do modelo.

Embora a acurácia seja uma métrica familiar e intuitiva, aprendemos por que talvez ela não forneça uma avaliação útil do desempenho de um classificador. Aprendemos como usar um modelo nulo de classe majoritária para saber se uma taxa de acurácia é realmente boa ou se não é melhor do que o resultado que obteríamos da previsão da classe mais comum de todas as amostras. Quando os dados estão desbalanceados, geralmente a acurácia não é a melhor maneira de julgar um classificador.

Para ter uma visão mais nuançada de como um modelo está se saindo, é necessário separar as classes positivas e negativas e avaliar sua acurácia independentemente. Das contagens resultantes de classificações de verdadeiros e falsos positivos e negativos, que podem ser resumidas em uma matriz de confusão, podemos derivar várias outras métricas: as taxas de verdadeiros e falsos positivos e negativos. Combinando os verdadeiros e falsos positivos e negativos com o conceito de probabilidades previstas e um limite variável de previsão, podemos caracterizar melhor a utilidade de um classificador usando a curva ROC, a curva precision-recall e as áreas sob essas curvas.

Com essas ferramentas, você está bem equipado para responder a perguntas gerais sobre o desempenho de um classificador binário, em qualquer área em que esteja trabalhando. Posteriormente no livro, conheceremos maneiras específicas de aplicações de avaliar o desempenho do modelo, anexando custos e benefícios a verdadeiros e falsos positivos e negativos. Antes disso, a partir do próximo capítulo, começaremos a aprender os detalhes existentes por trás daquele que possivelmente é o mais popular e simples modelo de classificação: a **regressão logística**.

Detalhes da regressão logística e exploração de características

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Criar list comprehensions em Python
- Descrever como funciona a regressão logística
- Formular as versões sigmóide e logit da regressão logística
- Utilizar a seleção univariada para encontrar características importantes
- Personalizar plotagens com a API Matplotlib
- Definir o limite de decisão linear de uma regressão logística

Este capítulo apresentará os aspectos básicos da regressão logística e vários outros métodos para verificarmos o relacionamento entre as características e a variável de resposta.

Introdução

No capítulo anterior, terminamos nossa investigação da variável de resposta e desenvolvemos alguns exemplos de modelos de machine learning usando o scikit-learn. No entanto, as características que usamos, EDUCATION e LIMIT_BAL, não foram selecionadas de maneira sistemática.

Neste capítulo, começaremos a desenvolver técnicas para a avaliação individual de cada característica. Isso nos permitirá percorrer rapidamente todas as características para ver quais podem ser úteis na modelagem preditiva. Para as características mais promissoras, veremos como criar resumos visuais que sirvam como ferramentas de

comunicação úteis.

Em seguida, iniciaremos nosso exame detalhado da regressão logística. Aprenderemos por que ela é considerada um modelo linear, mesmo quando as informações envolvem funções não lineares. Como consequência importante dessa linearidade, veremos por que o limite de decisão pode dificultar a classificação precisa dos dados. Conforme avançarmos, aprenderemos como escrever funções em Python.

Examinando os relacionamentos entre as características e a resposta

Para fazermos previsões precisas da variável de resposta, boas características são necessárias. Precisamos de características que estejam de alguma forma claramente vinculadas à variável de resposta. Até agora, examinamos o relacionamento entre algumas características e a variável de resposta, calculando o resultado da operação `groupby/mean` da variável, ou testando modelos diretamente, que é outra maneira de fazer esse tipo de exploração. Porém, ainda não fizemos uma exploração sistemática de como todas as características estão relacionadas à variável de resposta. É o que faremos agora, beneficiando-nos do esforço árduo investido quando exploramos as características para nos certificar se a qualidade dos dados era boa.

Uma maneira popular de examinar rapidamente como todas as características estão relacionadas à variável de resposta, e como elas se relacionam umas com as outras, é usando uma **plotagem de correlação**. Primeiro criaremos a plotagem de correlação para os dados do estudo de caso e depois discutiremos como interpretá-la e veremos alguns detalhes matemáticos.

Para criarmos uma plotagem de correlação, as entradas necessárias devem incluir todas as características que serão exploradas, além da variável de resposta. Já que usaremos a maioria dos nomes de colunas do `DataFrame` para fazer isso, uma maneira rápida de obter a lista apropriada em Python é começar com todos os nomes e remover os que não quisermos da lista. Como etapa preliminar, iniciaremos um novo notebook para este capítulo e carregaremos os pacotes e os dados limpos do *Capítulo 1, Exploração e limpeza de dados*, com este código:

```
import numpy as np #cálculo numérico
```

```
import pandas as pd #preparação dos dados
import matplotlib.pyplot as plt #pacote de plotagem
#A próxima linha ajuda a renderizar plotagens
%matplotlib inline
import matplotlib as mpl #adiciona a funcionalidade de plotagem
import seaborn as sns #um pacote de plotagem interessante
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Nota: O caminho de seu arquivo de dados limpos pode ser diferente, dependendo de onde você o salvou no *Cap 1, Exploração e limpeza de dados*.

Observe que esse notebook começa de maneira semelhante ao do capítulo anterior, exceto por também estarmos importando o pacote **Seaborn**, que tem muitos recursos de plotagem convenientes baseados no **Matplotlib**. Agora faremos uma lista de todas as colunas do **DataFrame** e examinaremos as primeiras e as últimas cinco:

```
features_response = df.columns.tolist()

features_response[:5]

['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE']

features_response[-5:]

['graduate school', 'high school', 'none', 'others', 'university']
```

Figura 3.1 – Obtenha uma lista com os nomes das colunas.

Lembre-se de que não devemos usar a variável de gênero devido a questões éticas e de que verificamos que PAY_2, PAY_3, ..., PAY_6 estão incorretas e devem ser ignoradas. Também não examinaremos a codificação one-hot que criamos a partir da variável EDUCATION, já que as informações dessas colunas já estão incluídas, pelo menos de certa forma, na característica original. Usaremos apenas a característica EDUCATION diretamente. Para concluir, não faz sentido usar ID como característica, porque se trata somente de um identificador de conta exclusivo que não tem ligação com a variável de resposta. Criaremos outra lista de nomes de colunas que não sejam características ou a resposta. Queremos excluir esses itens de nossa análise:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school', 'none',
                   'others', 'university']
```

Para termos uma lista de nomes de colunas composta somente das características e da resposta que usaremos, removeremos os nomes

contidos em `items_to_remove` da lista atual existente em `features_response`. Há várias maneiras de fazer isso em Python. Aproveitaremos essa oportunidade para conhecer uma maneira específica de construir uma lista em Python, chamada **list comprehension**. Quando as pessoas se referem a certas estruturas como sendo **Pythônicas**, ou pertencentes ao idioma da linguagem Python, com frequência as list comprehensions são mencionadas.

O que é uma list comprehension? Conceitualmente, é quase a mesma coisa que um loop for. No entanto, as list comprehensions permitem que a criação de listas, que seriam distribuídas em várias linhas de um loop for real, seja escrita em uma única linha. Elas também são um pouco mais rápidas que os loops for, devido às otimizações que ocorrem em Python. Provavelmente isso não vai economizar muito tempo aqui, mas é uma boa chance de nos familiarizarmos com elas. A seguir temos um exemplo de list comprehension:

```
example_list_comp = [item for item in range(5)]
example_list_comp

[0, 1, 2, 3, 4]
```

Figura 3.2 – Exemplo de uma list comprehension.

É só isso!

Também podemos usar cláusulas tradicionais para tornar as list comprehensions flexíveis. Por exemplo, podemos usá-las para reatribuir a variável `features_response` com uma lista contendo tudo que não esteja na lista de strings que queremos remover:

```
features_response = [item for item in features_response if item not in items_to_remove]
features_response

['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

Figura 3.3 – Usando uma list comprehension para remover nomes de colunas.

O uso de `if` e `not in` na list comprehension é autoexplicativo. A fácil

legibilidade de estruturas como as list comprehensions é uma das razões para a popularidade de Python.

Nota: A documentação da linguagem Python (<https://docs.python.org/3/tutorial/datastructures.html>) define as list comprehensions da seguinte forma:

“Uma list comprehension é composta de colchetes que contêm uma expressão seguida por uma cláusula for e depois zero ou mais cláusulas for ou if”

Logo, as list comprehensions nos permitem fazer o que queremos com menos código, de um jeito que geralmente é bastante legível e compreensível.

Correlação de Pearson

Estamos prontos para criar nossa plotagem de correlação. Ela sempre se baseia em uma **matriz de correlação**, que devemos calcular primeiro. O pandas facilita isso. Só temos de selecionar nossas colunas de características e valores de resposta usando a lista que acabamos de criar e chamar o método `.corr()` nelas. Enquanto fazemos o cálculo, observe que o tipo de correlação que está disponível para nós no pandas é a **correlação linear**, também conhecida como **correlação de Pearson**. A correlação de Pearson é usada para medir a força e a direção (isto é, positiva ou negativa) do relacionamento linear entre duas variáveis:

```
corr = df[features_response].corr()
corr.iloc[0:5,0:5]
```

	LIMIT_BAL	EDUCATION	MARRIAGE	AGE	PAY_1
LIMIT_BAL	1.000000	-0.232688	-0.111873	0.149157	-0.273396
EDUCATION	-0.232688	1.000000	-0.137097	0.179035	0.112653
MARRIAGE	-0.111873	-0.137097	1.000000	-0.412828	0.019759
AGE	0.149157	0.179035	-0.412828	1.000000	-0.044277
PAY_1	-0.273396	0.112653	0.019759	-0.044277	1.000000

Figura 3.4 – Cinco primeiras linhas e colunas da matriz de correlação.

Após criar a matriz de correlação, você notará que os nomes das linhas e colunas são iguais. Em seguida, para cada comparação possível entre todos os pares de características, assim como para todas as características e a resposta, que ainda não podemos ver aqui nas cinco primeiras linhas e colunas, há um número. Ele é chamado de **correlação** entre essas duas colunas. Você deve ter notado que todas

as correlações estão entre -1 e 1; uma coluna tem correlação igual a 1 consigo mesma (a diagonal da matriz de correlação) e há repetição: cada comparação aparece duas vezes já que cada nome de coluna do DataFrame original aparece tanto como uma linha quanto como uma coluna na matriz de correlação. Antes de aprender mais sobre a correlação, usaremos o Seaborn para criar uma plotagem refinada. Esse é o código da plotagem, seguido pela saída:

```
sns.heatmap(corr,  
            xticklabels=corr.columns.values,  
            yticklabels=corr.columns.values,  
            center=0)
```

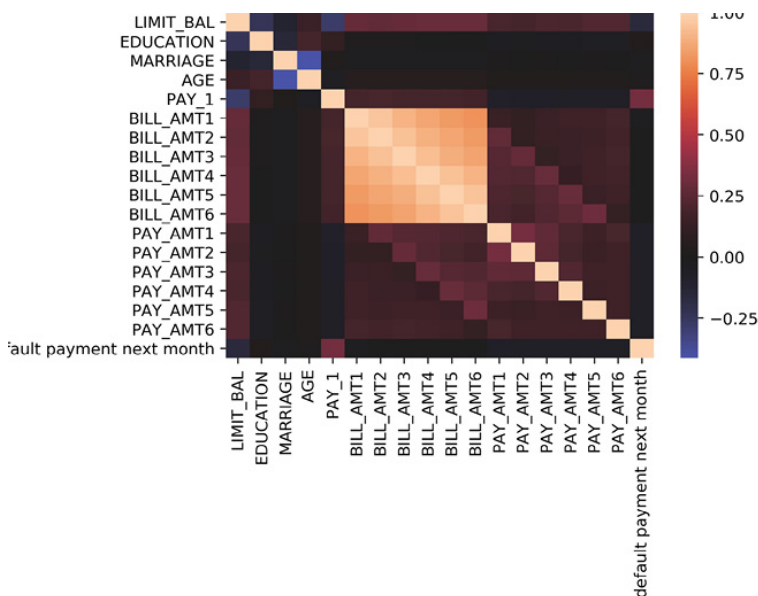


Figura 3.5 – Heatmap da plotagem de correlação no Seaborn.

O heatmap do Seaborn cria uma visualização clara da matriz de correlação, de acordo com a escala de cor que aparece à direita na *Figura 3.5*, chamada de barra de cores. Observe que ao chamar `sns.heatmap`, além da matriz, fornecemos os **tick labels** para os eixos x e y, que são as características e o nome da resposta, e indicamos que o centro da barra de cores deve ser em 0, para que as correlações positiva e negativa sejam distinguíveis como vermelha e azul, respectivamente.

O que essa plotagem mostra? Em um nível geral, se duas características, ou uma característica e a resposta, estiverem **altamente correlacionadas**, podemos dizer que há uma forte associação entre elas. As características que estiverem altamente

correlacionadas com a resposta serão úteis na previsão. Essa alta correlação pode ser positiva ou negativa; explicaremos a diferença em breve.

A plotagem de correlação mostra, ao longo da última linha (ou da última coluna), que a característica PAY_1 é a mais fortemente correlacionada com a variável de resposta. Também podemos ver que várias características estão altamente correlacionadas umas com as outras, principalmente as características BILL_AMT. No próximo capítulo, falaremos sobre a importância das características que estão correlacionadas; essa informação é relevante para certos modelos como o de regressão logística, que faz suposições sobre as correlações entre características. Por enquanto, consideraremos a observação de que PAY_1 provavelmente será a característica com melhores resultados de previsão para nosso modelo. A outra característica que parece ser importante é LIMIT_BAL, que tem correlação negativa. Se você tem boa visão, perceberá que só essas duas características parecem ter uma cor diferente de preto (que indica correlação 0) na última linha da *Figura 3.5*.

O que é correlação linear, matematicamente falando? Se você conhece estatística básica, deve estar familiarizado com a correlação linear. Para duas colunas, X e Y , a correlação linear ρ (letra grega minúscula “rô”) é definida da seguinte forma:

$$\rho = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y}$$

Figura 3.6 – Equação da correlação linear.

Essa equação descreve o **valor esperado** (E , que podemos considerar como a média) da diferença entre os elementos de X , e sua média, μ_X , multiplicada pela diferença entre os elementos correspondentes de Y , e sua média, μ_Y . A média é obtida com pares de valores X , Y . Quando X é relativamente alto (comparado com sua média), Y também tende a ser alto, e da mesma forma Y tende a ser baixo quando X é baixo, então, os termos da multiplicação no numerador tenderão a ser ambos positivos ou ambos negativos, levando a um produto positivo e a uma **correlação positiva** após o valor esperado ser calculado. Por outro lado, se Y diminuir enquanto X aumenta, eles terão **correlação negativa**. O denominador (produto dos **desvios padrão** de X e Y) serve para normalizar a correlação linear para a escala de $[-1, 1]$. Já que a correlação de Pearson é ajustada para a média e o desvio padrão dos dados, os valores reais não são tão importantes quanto o

relacionamento entre X e Y . As correlações lineares mais fortes ficam próximas de 1 ou -1. Se não houver relação linear entre X e Y , a correlação ficará perto de zero.

É bom lembrar que, embora seja usada regularmente nesse contexto pelos praticantes da ciência de dados, a correlação de Pearson não é muito apropriada para uma variável de resposta binária, como a que temos em nosso problema. Tecnicamente falando, entre outras restrições, ela só é válida para **dados contínuos**, como os usados no exercício de regressão linear do *Capítulo 2, Introdução ao scikit-learn e avaliação do modelo*. No entanto, pode ajudar a dar uma visão geral da utilidade das características. A correlação de Pearson também está convenientemente disponível em bibliotecas como o pandas.

Você perceberá que na ciência de dados certas técnicas amplamente usadas são aplicadas a dados que violam suas suposições estatísticas formais. É importante conhecer as suposições formais que os métodos fazem. Na verdade, o conhecimento dessas suposições pode ser avaliado durante entrevistas para cargos de ciência de dados. Porém, na prática, se a técnica nos ajudar a entender o problema e encontrar uma solução eficaz, será uma ferramenta valiosa.

Dito isso, a correlação linear não é uma medida eficiente para o poder preditivo de todas as características. Ela só funciona em relacionamentos lineares. Mudando um pouco nosso foco para um problema de regressão hipotético, examine os exemplos a seguir e pense em qual seriam as correlações lineares. Observe que os valores dos dados nos eixos x e y não foram rotulados; isso está ocorrendo porque a localização (média) e o desvio padrão (escala) dos dados não afetam a correlação de Pearson; ela só é afetada pelo relacionamento entre os dados, que podemos distinguir ao plotá-los.

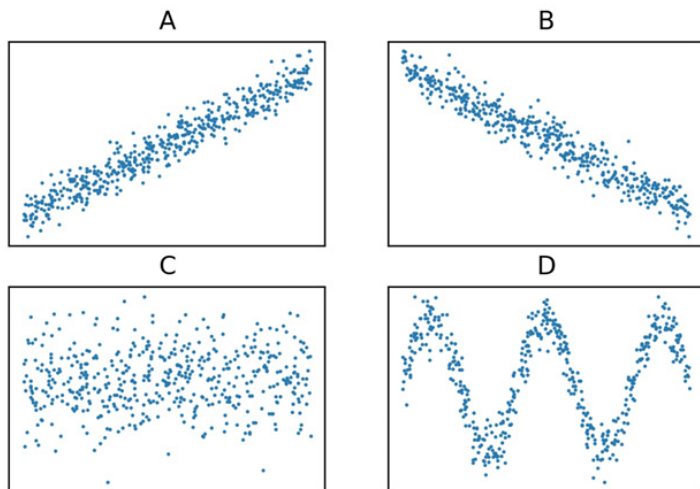


Figura 3.7 – Gráfico de dispersão da relação entre exemplos de variáveis.

Nos exemplos A e B, as correlações de Pearson para os datasets são 0.96 e -0.97, respectivamente, de acordo com a fórmula já fornecida. Ao examinarmos as plotagens, fica claro que uma correlação perto de 1 ou -1 fornece um insight útil sobre o relacionamento entre essas variáveis. Para o exemplo C, a correlação é de 0.06. Uma correlação próxima a 0 parece uma indicação óbvia da falta de associação: o valor de Y não parece ter muito a ver com o valor de X. Contudo, no exemplo D, há claramente algum relacionamento entre as variáveis. No entanto, a correlação linear é mais baixa que no exemplo anterior, ficando em 0.02! Nesse caso, X e Y tendem a “moverem-se juntos” em escalas menores, mas a média é obtida considerando-se todas as amostras quando a correlação linear é calculada.

No fim das contas, qualquer síntese estatística, como a correlação, que você usar será somente isso: uma síntese. Ela pode ocultar detalhes importantes. Logo, em geral é uma boa ideia examinar visualmente o relacionamento entre as características e a resposta. Isso ocuparia muito espaço na página, portanto, não demonstraremos para todas as características no estudo de caso. Porém, tanto o pandas quanto o Seaborn oferecem funções para a criação do que é chamado de **matriz de gráfico de dispersão**. Essa matriz é semelhante a uma plotagem de correlação, mas exibe todos os dados como uma grade de gráficos de dispersão de todas as características e da variável de resposta. Isso permite examinar os dados diretamente em um formato conciso. Já que pode envolver muitos dados e plotagens, talvez seja necessário usar amostras de dados menores e examinar um número reduzido de características para a função ser executada eficientemente.

Teste F

Embora teoricamente a correlação de Pearson seja válida para variáveis de resposta contínuas, a variável de resposta binária dos dados do estudo de caso pode ser considerada um dado categórico com apenas duas categorias: 0 e 1. Entre os diferentes tipos de teste que podemos executar para ver se as características estão associadas a uma resposta categórica, temos o **teste F ANOVA**, disponível no scikit-learn como `f_classif`. ANOVA é a abreviação de “analysis of variance” (análise de variância). O teste F ANOVA pode ser comparado com o **teste F de regressão**, que é muito semelhante à correlação de Pearson, e também está disponível no scikit-learn como `f_regression`.

Executaremos um teste F ANOVA usando as características candidatas dos dados do estudo de caso no exercício a seguir. Você verá que a saída é composta da estatística f e de valores-p. Como interpretá-la? Vamos nos concentrar no valor-p, por razões que ficarão claras no exercício. O valor-p é um conceito útil para várias medidas estatísticas. Por exemplo, embora não as tenhamos examinado, cada uma das correlações de Pearson calculadas para a matriz de correlação anterior tem um valor-p correspondente. Há um conceito de valor-p semelhante aplicável aos coeficientes da regressão linear, aos coeficientes da regressão logística e a outras medidas.

No contexto do teste F, o valor-p responde à seguinte pergunta: “Qual é a probabilidade de o valor da média dessa característica nas amostras da classe positiva ser igual ao de amostras da classe negativa?”. Se uma característica tiver valores de média muito diferentes entre as classes positiva e negativa:

- É improvável que os valores da média sejam iguais (valor-p baixo)
- Provavelmente será uma boa característica em nosso modelo porque nos ajudará a distinguir as classes positivas das negativas

Mantenha isso em mente durante o exercício a seguir.

Exercício 11: Teste F e seleção de características univariada

Neste exercício, usaremos o teste F para examinar a relação entre as características e a resposta. Investigaremos esse método como parte do que é chamado de **seleção de características univariada**: a prática de testar as características uma a uma em relação à variável de

resposta para ver quais têm poder preditivo. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída dos Exercícios 11-15 e da Atividade 3 foram carregados em um Jupyter notebook que pode ser encontrado aqui: <http://bit.ly/2Dz5iNA>. Você pode rolar até a seção apropriada dentro do Jupyter notebook para localizar o exercício ou a atividade que procura.

1. A primeira etapa para a execução do teste F ANOVA é separar as características e a resposta como arrays NumPy usando a lista que criamos e a indexação de inteiros do pandas:

```
X = df[features_response].iloc[:, :-1].values
y = df[features_response].iloc[:, :-1].values
print(X.shape, y.shape)
```

A saída deve exibir as dimensões das características e da resposta:

```
X = df[features_response].iloc[:, :-1].values
y = df[features_response].iloc[:, :-1].values
print(X.shape, y.shape)

(26664, 17) (26664,)

from sklearn.feature_selection import f_classif

[f_stat, f_p_value] = f_classif(X, y)
```

Figura 3.8 – Dimensão dos arrays de características e da resposta.

Há 17 características, e os arrays tanto de características quanto da resposta têm o mesmo número de amostras como esperado.

2. Importe a função `f_classif` e forneça as características e a resposta:

```
from sklearn.feature_selection import f_classif
[f_stat, f_p_value] = f_classif(X, y)
```

Dois saídas são obtidas de `f_classif`: a **estatística F** e o **valor-p**, para a comparação de cada característica com a variável de resposta. Criaremos um novo DataFrame contendo os nomes das características e essas saídas para facilitar nossa inspeção. Uma maneira de especificar um novo DataFrame é usando um **dicionário**, com pares **chave:valor** de nomes de colunas e dos dados que ficarão contidos em cada uma delas. Mostraremos o DataFrame classificado (de forma ascendente) pelo valor-p.

3. Use este código para criar um DataFrame com nomes de características, com a estatística F e com os valores-p e exiba-o classificado pelo valor-p:

```
f_test_df = pd.DataFrame({'Feature': features_response[:, :-1],
```

```

        'F statistic':f_stat,
        'p value':f_p_value})
f_test_df.sort_values('p value')

```

A saída deve ser esta:

```

f_test_df = pd.DataFrame({'Feature':features_response[:-1],
                           'F statistic':f_stat,
                           'p value':f_p_value})
f_test_df.sort_values('p value')

```

	Feature	F statistic	p value
4	PAY_1	3156.672300	0.000000e+00
0	LIMIT_BAL	651.324071	5.838366e-142
11	PAY_AMT1	140.612679	2.358354e-32
12	PAY_AMT2	101.408321	8.256124e-24
13	PAY_AMT3	90.023873	2.542641e-21
15	PAY_AMT5	85.843295	2.090120e-20
16	PAY_AMT6	80.420784	3.219565e-19
14	PAY_AMT4	79.640021	4.774112e-19
1	EDUCATION	32.637768	1.122175e-08
2	MARRIAGE	18.078027	2.127555e-05
5	BILL_AMT1	11.218406	8.110226e-04
7	BILL_AMT3	5.722938	1.675157e-02
6	BILL_AMT2	5.668454	1.727965e-02
3	AGE	5.479140	1.925206e-02
8	BILL_AMT4	3.434740	6.384965e-02
9	BILL_AMT5	1.216082	2.701409e-01
10	BILL_AMT6	1.049561	3.056176e-01

Figura 3.9 – Resultados do teste F ANOVA.

Observe que, para cada diminuição no valor-p, há um aumento na estatística F, logo, as informações dessas colunas são basicamente as mesmas em termos de classificação de características.

As conclusões a que podemos chegar pelo DataFrame da estatística F e dos valores-p são semelhantes ao que observamos na plotagem de correlação: PAY_1 e LIMIT_BAL parecem ser as características mais úteis. Elas têm valores-p menores, o que indica que os valores da média dessas características entre a classe positiva e negativa são **significativamente diferentes**, e ajudarão a prever a que classe uma amostra pertence.

No scikit-learn, um dos usos para avaliações como o teste F é na execução da **seleção de características univariada**. Isso pode ser benéfico se você tiver um número muito grande de características, várias talvez totalmente inúteis, e quiser uma maneira rápida de

obter uma “lista curta” com as mais úteis. Por exemplo, se quiséssemos recuperar apenas 20% das características com maior estatística F, poderíamos fazer isso facilmente com a classe `SelectPercentile`. É bom ressaltar também que há uma classe semelhante para a seleção das “k” principais características (em que k é qualquer número que especificarmos), chamada `SelectKBest`. Aqui demonstraremos como selecionar as 20% melhores.

4. Para selecionar as características consideradas pelo teste F como as 20% melhores, primeiro importe a classe `SelectPercentile`:

```
from sklearn.feature_selection import SelectPercentile
```

5. Instancie um objeto dessa classe, indicando que queremos usar o mesmo critério de seleção de características, o teste F ANOVA, que consideramos até agora neste exercício, e que, entre as características, queremos as 20% melhores.

```
selector = SelectPercentile(f_classif, percentile=20)
```

6. Use o método `.fit` para ajustar o objeto às características e aos dados da resposta, semelhante a como um modelo seria ajustado:

```
selector.fit(X, y)
```

A saída pode ser visualizada na Figura 3.10.

Há várias maneiras de acessar as características selecionadas diretamente, o que você pode aprender na documentação do scikit-learn (examinando o método `.transform`, ou o emprego de `.fit_transform` na etapa do ajuste). No entanto, esses métodos retornarão arrays NumPy, que não informam os nomes das características que foram selecionadas, somente os valores. Para resolver isso, você pode usar o método `.get_support` do objeto seletor de características, que lhe informará os índices de colunas selecionados no array de características.

```
from sklearn.feature_selection import SelectPercentile

selector = SelectPercentile(f_classif, percentile=20)

selector.fit(X, y)

SelectPercentile(percentile=20,
                  score_func=<function f_classif at 0x1a218ba730>)
```

Figura 3.10 – Seleção de características univariada no scikit-learn buscando as 20% melhores.

7. Capture os índices das características selecionadas em um array

chamado `best_feature_ix`:

```
best_feature_ix = selector.get_support()
```

```
best_feature_ix
```

A saída deve ser igual à mostrada a seguir, indicando um índice lógico que pode ser usado com um array de nomes de características, assim como valores, supondo que estejam na mesma ordem do array de características fornecido para `SelectPercentile`:

```
best_feature_ix = selector.get_support()
best_feature_ix

array([ True, False, False, False,  True, False, False, False, False,
        False, False,  True,  True, False, False, False, False])

features = features_response[:-1]

best_features = [features[counter] for counter in range(len(features))
                  if best_feature_ix[counter]]

best_features

['LIMIT_BAL', 'PAY_1', 'PAY_AMT1', 'PAY_AMT2']
```

Figura 3.11 – Índice lógico das características selecionadas.

8. Os nomes das características podem ser obtidos com o uso de todos os elementos exceto o último (nome da variável de resposta) de nossa lista `features_response` pela indexação com `:-1`:

```
features = features_response[:-1]
```

9. Use o array de índices que criamos na *Etapa 7* com uma list comprehension e a lista `features` para encontrar os nomes das características selecionadas, como descrito a seguir:

```
best_features = [features[counter] for counter in range(len(features))
                  if best_feature_ix[counter]]

best_features
```

A saída deve ser a seguinte:

```
best_feature_ix = selector.get_support()
best_feature_ix

array([ True, False, False, False,  True, False, False, False, False,
        False, False,  True,  True, False, False, False, False])

features = features_response[:-1]

best_features = [features[counter] for counter in range(len(features))
                  if best_feature_ix[counter]]

best_features

['LIMIT_BAL', 'PAY_1', 'PAY_AMT1', 'PAY_AMT2']
```

Figura 3.12 – Examinando os rótulos das características que são as 20% melhores.

Nesse código, a list comprehension percorreu os elementos do array `features` (`len(features)`) com o incremento de `loop counter`, usando o array booleano `best_feature_ix`, que representa as características selecionadas, na instrução `if` para verificar cada característica e capturar o nome das escolhidas.

As características selecionadas equivalem às quatro primeiras linhas do DataFrame de resultados do teste F, logo, a seleção funcionou como esperado. Embora não seja necessário agir das duas maneiras, já que ambas levam ao mesmo resultado, é bom fazê-lo para avaliar o trabalho, considerando-se que você está aprendendo novos conceitos. É preciso estar alerta para o fato de que, com métodos de conveniência como `SelectPercentile`, você não visualizará a estatística F ou os valores-p. No entanto, em algumas situações pode ser melhor usá-los porque os valores-p podem não ser importantes a não ser na classificação de características.

Pontos mais importantes do teste F: equivalência com o teste t para duas classes e cuidados

Quando usamos um teste F para examinar a diferença de média somente entre dois grupos, como fizemos aqui para o problema de classificação binária do estudo de caso, na verdade o teste se resume ao que é conhecido como **teste t**. Um teste F é extensível a três ou mais grupos e, portanto, é útil para a classificação multiclasse. Um teste t compara a média apenas entre dois grupos de amostras para ver se a diferença é **estatisticamente significativa**.

Embora no nosso caso o teste F tenha sido útil para a seleção de características univariada, é preciso tomar alguns cuidados. Voltando ao conceito das suposições estatísticas formais, no teste F elas incluem a de que os dados estão **normalmente distribuídos**. Não verificamos essa suposição. Além disso, na comparação da mesma variável de resposta, `y`, com possíveis características da matriz, `X`, executamos o que é conhecido em estatística como **comparações múltiplas**. Resumindo, isso significa que, examinando várias características e comparando-as repetidamente com a mesma resposta, aumentam as chances de encontrarmos aleatoriamente uma “boa característica”. Porém, talvez essas características não possam ser generalizadas para novos dados. As **correções estatísticas para comparações múltiplas** ajustam os valores-p para resolver esse problema.

Mesmo se não seguirmos todas as “regras” estatísticas exigidas por

esses métodos, podemos obter resultados úteis. Muitos métodos que assumem uma distribuição normal são usados regularmente com dados não normais, com frequência com resultados aceitáveis. E a correção da comparação múltipla é uma preocupação maior quando os valores-p são os que mais interessam, por exemplo, ao fazermos inferências estatísticas. Aqui, os valores-p são apenas o meio para chegarmos ao objetivo de classificar a lista de características. A ordem dessa classificação não mudaria se os valores-p fossem corrigidos para comparações múltiplas.

Além de saber que características podem ser úteis na modelagem, é bom conhecermos melhor as características mais importantes. Portanto, vamos explorá-las graficamente e com detalhes no próximo exercício. Posteriormente examinaremos outros métodos de seleção de características que não fazem as mesmas suposições que apresentamos aqui e que terão uma integração mais direta com os modelos preditivos que construiremos.

Hipóteses e próximas etapas

De acordo com nossa exploração de características univariada, a característica de maior associação com a variável de resposta é PAY_1. Isso faz sentido? O que significa PAY_1? É o status de pagamento da conta no mês mais recente. Como aprendemos na exploração de dados inicial, há alguns valores que indicam que a conta está em boa situação: -2 significa que ela não foi usada, -1 indica saldo totalmente quitado e 0 significa que pelo menos o pagamento mínimo foi feito. Por outro lado, valores inteiros positivos indicam atraso no pagamento relativo a esse número de meses. As contas com pagamentos atrasados no último mês podem ser consideradas inadimplentes. Isso significa basicamente que essa característica captura valores históricos da variável de resposta. Características como essa são muito importantes como *um dos melhores preditores para qualquer problema de machine learning que use dados históricos para a mesma coisa que estamos tentando prever (ou seja, a variável de resposta)*. Parece lógico: pessoas que já ficaram inadimplentes apresentam maior risco de inadimplir novamente.

E quanto a LIMIT_BAL, o limite de crédito das contas? Se considerarmos como os limites de crédito são atribuídos, é provável que nosso cliente tenha avaliado o risco que um tomador de empréstimo pode apresentar ao decidir sobre o limite. Clientes que apresentam mais risco devem receber limites mais baixos para que o

credor tenha menos problemas. Logo, podemos esperar uma maior probabilidade de inadimplência para contas com valores menores para `LIMIT_BAL`.

O que aprendemos com nosso exercício de seleção de características univariada? Temos uma ideia de quais devem ser as características mais importantes de nosso modelo. E, a partir da matriz de correlação, temos uma noção de como elas estão relacionadas com a variável de resposta. No entanto, conhecendo as limitações dos testes que usamos, é uma boa ideia visualizar essas características para obtermos um quadro mais detalhado da relação entre elas e a resposta. Também começamos a desenvolver **hipóteses** sobre as características: por que as achamos importantes? Visualizando os relacionamentos entre as características e a variável de resposta, poderemos determinar se nossas ideias são compatíveis com o que vimos nos dados.

As hipóteses e visualizações costumam ser uma parte importante da apresentação de resultados para o cliente, que pode estar interessado em saber como o modelo funciona, e não apenas no fato de que ele funciona.

Exercício 12: Visualizando o relacionamento entre as características e a resposta

Neste exercício, você aumentará seu conhecimento sobre as funções de plotagem do Matplotlib que usamos no livro. Aprenderemos como personalizar os gráficos para responder melhor a perguntas específicas usando os dados. Ao fazer essas análises, criaremos visualizações significativas de como as características `PAY_1` e `LIMIT_BAL` estão relacionadas com a variável de resposta, o que pode dar suporte às hipóteses que formulamos sobre elas. Isso será feito pela aquisição de um conhecimento maior da **API** Matplotlib. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter notebook que pode ser encontrado aqui: <http://bit.ly/2Dz5iNA>.

1. Calcule a linha de base da variável de resposta para a taxa de inadimplência de todo o dataset usando o método `.mean()` do pandas:

```
overall_default_rate = df['default payment next month'].mean()
overall_default_rate
```

A saída deve ser esta:

0.2217971797179718

Figura 3.13 – Taxa de inadimplência de todo o dataset.

Qual seria uma boa maneira de visualizar as taxas de inadimplência para diferentes valores da característica PAY_1?

Lembre-se de que verificamos que essa característica é híbrida, sendo tanto categórica quanto numérica. Vamos plotá-la de uma forma que é típica de características categóricas, devido ao número relativamente pequeno de valores exclusivos. No *Capítulo 1, Exploração e limpeza de dados* fizemos contagens de valores dessa característica como parte da exploração de dados e depois conhecemos groupby/mean quando examinamos a característica EDUCATION. groupby/mean seria uma boa maneira de visualizarmos a taxa de inadimplência para diferentes status de pagamento.

2. Use este código para criar uma agregação groupby/mean:

```
group_by_pay_mean_y = df.groupby('PAY_1').agg({'default payment next month':np.mean})  
group_by_pay_mean_y
```

A saída pode ser visualizada na Figura 3.14.

```
group_by_pay_mean_y = df.groupby('PAY_1').agg({'default payment next month':np.mean})  
group_by_pay_mean_y
```

default payment next month	
PAY_1	
-2	0.131664
-1	0.170002
0	0.128295
1	0.336400
2	0.694701
3	0.773973
4	0.682540
5	0.434783
6	0.545455
7	0.777778
8	0.588235

Figura 3.14 – Média da variável de resposta por grupos da característica PAY_1.

Com a verificação desses valores já podemos discernir a tendência. Passaremos direto à sua plotagem. Vamos executá-la por etapas e introduziremos alguns conceitos novos. Você deve inserir todo o

código das etapas 3 a 6 na mesma célula.

No Matplotlib, cada plotagem aparece em um objeto `axes` e dentro de uma janela `figure`. Criando objetos para `axes` e `figures`, você pode acessar e alterar diretamente suas propriedades, inclusive os rótulos dos eixos, as tick marks, e outros elementos dos eixos, ou as dimensões da figura.

3. Crie um objeto `axes` em uma variável também chamada `axes` usando o código a seguir:

```
axes = plt.axes()
```

4. Plote a taxa de inadimplência geral como uma linha horizontal vermelha.

O Matplotlib facilita essa operação; você só precisa indicar a interceptação de `y` na linha com a função `axhline`. Observe que, em vez de chamar a função a partir de `plt`, estamos chamando-a como um método em nosso objeto `axes`:

```
axes.axhline(overall_default_rate, color='red')
```

Agora, sobre essa linha, queremos plotar a taxa de inadimplência dentro de cada grupo de valores de `PAY_1`.

5. Use o método `plot` do `DataFrame` de dados agrupados que criamos. Especifique um marcador `'x'` ao longo da plotagem da linha e a inexistência de uma instância de `legend`, que criaremos posteriormente; informe que o **eixo pai** da plotagem deve ser o objeto `axes` com o qual estamos trabalhando (caso contrário, o `pandas` removerá o objeto existente e criará um novo objeto `axes`):

```
group_by_pay_mean_y.plot(marker='x', legend=False, ax=axes)
```

Esses são os dados que queremos plotar.

6. Defina o rótulo do eixo `y` e crie uma instância de `legend` (há muitas opções para controlarmos a aparência da legenda, mas uma maneira simples é fornecer uma lista de strings indicando os rótulos dos elementos gráficos na ordem em que foram adicionados a `axes`):

```
axes.set_ylabel('Proportion of credit defaults')  
axes.legend(['Entire dataset', 'Groups of PAY_1'])
```

7. A execução de todo o código das *Etapas 3 a 6* em uma única célula deve resultar na plotagem a seguir:

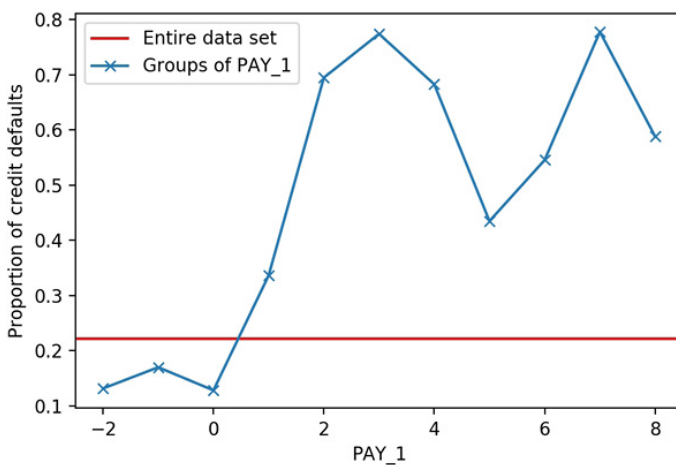


Figura 3.15 – Taxas de não pagamento de dívidas para todos os dados.

Nossa visualização dos status de pagamento revelou uma história clara e provavelmente esperada: quem já inadimpliu apresenta uma tendência maior de fazê-lo novamente. A taxa de inadimplência de contas em boa situação está bem abaixo da taxa geral, que sabemos que era de aproximadamente 22%. No entanto, pelo menos 30% das contas que estavam inadimplentes no último mês estarão inadimplentes novamente no próximo mês de acordo com a verificação. Essa é uma boa referência visual para compartilharmos com o sócio da empresa já que mostra o efeito da característica que talvez seja a mais importante de nosso modelo.

Agora voltaremos nossa atenção para a característica classificada como tendo a segunda associação mais forte com a variável alvo: LIMIT_BAL. Trata-se de uma característica numérica com muitos valores exclusivos. Uma boa maneira de visualizar características como essa, para um problema de classificação, é plotar vários histogramas no mesmo eixo, com diferentes cores para classes distintas. Como maneira de separar as classes, podemos indexá-las a partir do DataFrame usando arrays lógicos.

8. Use este código para criar máscaras lógicas para amostras positivas e negativas:

```
pos_mask = y == 1
neg_mask = y == 0
```

Para criar nossa plotagem de histograma dupla, geraremos outro objeto axes e chamaremos o método .hist nele duas vezes para os histogramas das classes positiva e negativa. Forneceremos alguns argumentos de palavra-chave adicionais: alpha cria transparência

nos histogramas, para que, se eles se sobrepuserem, ainda possamos vê-los, e especificaremos as cores. As cores azul e vermelha, com transparência, exibirão uma cor roxa em locais em que os histogramas se sobrepuserem. Uma vez que tivermos os histogramas, giraremos os tick labels do eixo x para torná-los mais legíveis e criaremos outras anotações que devem ser autoexplicativas.

9. Use o código a seguir para criar a plotagem de histograma dupla com as propriedades já mencionadas:

```
axes = plt.axes()
axes.hist(df.loc[neg_mask, 'LIMIT_BAL'], alpha=0.5, color='blue')
axes.hist(df.loc[pos_mask, 'LIMIT_BAL'], alpha=0.5, color='red')
axes.tick_params(axis='x', labelrotation=45)
axes.set_xlabel('Credit limit (NT$)')
axes.set_ylabel('Number of accounts')
axes.legend(['Not defaulted', 'Defaulted'])
axes.set_title('Credit limits by response variable')
```

A plotagem pode ser visualizada na Figura 3.16.

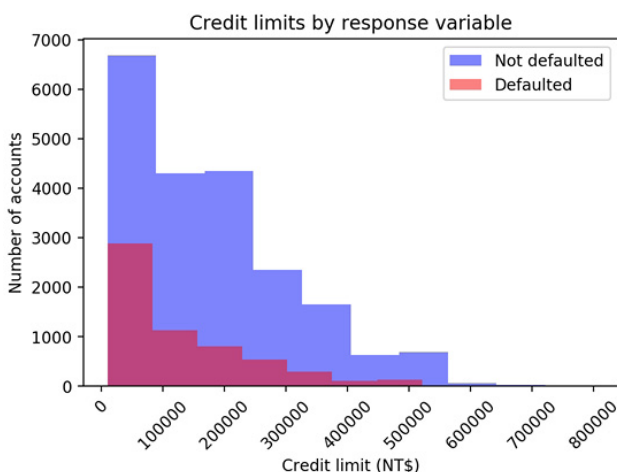


Figura 3.16 – Histogramas duplos dos limites de crédito.

Embora essa plotagem tenha ficado com a formatação desejada, ela não é tão explicativa como poderia. Queríamos olhar para ela e constatar que o limite de crédito pode ser uma boa maneira de diferenciar as contas inadimplentes das que não inadimplirão. No entanto, nossa principal percepção visual é a de que o histograma azul é maior que o vermelho. Isso ocorre porque há menos contas inadimplentes. Já sabíamos desse fato pela verificação das frações das classes.

Seria mais informativo exibir algo sobre como as formas desses histogramas são diferentes, e não apenas seus tamanhos. Para enfatizar esse ponto, podemos tornar igual a área plotada total dos dois histogramas, **normalizando-os**. O Matplotlib fornece um argumento de palavra-chave que facilita a criação do que poderíamos considerar a versão empírica de uma **função de densidade de probabilidade**. Isso significa que a integral, ou a área contida dentro de cada histograma, será igual a 1 após a normalização, já que as probabilidades resultam em 1.

Após alguns testes, decidimos criar um histograma com 16 bins. Já que o limite máximo de crédito é NT\$800,000, usamos range com um incremento de NT\$50000.

10. Crie as bordas dos bins do histograma com esse código, que também exibirá a borda de bin final como uma verificação:

```
bin_edges = list(range(0,850000,50000))  
print(bin_edges[-1])
```

A saída deve ser:

```
df[ 'LIMIT_BAL' ].max()  
  
800000  
  
bin_edges = list(range(0,850000,50000))  
print(bin_edges[-1])  
  
800000
```

Figura 3.17 – Observando o limite máximo de crédito e criando bordas de bins para histogramas normalizados.

O código de plotagem dos histogramas normalizados é semelhante ao anterior, com algumas alterações importantes: o uso da palavra-chave bins para definir a localização das bordas dos bins, density=True para normalizar os histogramas, e alterações nas anotações da plotagem. A parte mais complexa é termos de ajustar os **tick labels de y** para que as alturas dos bins do histograma tenham a interpretação das proporções, que é mais intuitiva que a saída padrão.

Os tick labels de y são os rótulos de texto exibidos perto dos ticks do eixo y e costumam ser simplesmente os valores dos ticks nesses locais. No entanto, você pode alterar isso manualmente se quiser.

Nota: De acordo com a documentação do Matplotlib, para termos um histograma normalizado, as alturas dos bins são calculadas com a “divisão da contagem pelo número de observações vezes

a largura do bin” (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.hist.html). Logo, temos de multiplicar os tick labels do eixo y pela largura do bin de NT\$50,000 para as alturas dos bins representarem a proporção do número total de amostras de cada bin. Observe as duas linhas em que estamos obtendo as localizações dos ticks do eixo y e definindo os rótulos para uma versão modificada. O arredondamento para duas casas decimais com `np.round` é necessário devido a pequenos erros da aritmética de ponto flutuante.

11. Execute este código para produzir histogramas normalizados:

```
mpl.rcParams['figure.dpi'] = 400
axes = plt.axes()
axes.hist(df.loc[neg_mask, 'LIMIT_BAL'], bins=bin_edges,
          alpha=0.5, density=True, color='blue')
axes.hist(df.loc[pos_mask, 'LIMIT_BAL'], bins=bin_edges,
          alpha=0.5, density=True, color='red')
axes.tick_params(axis='x', labelrotation=45)
axes.set_xlabel('Credit limit (NT$)')
axes.set_ylabel('Proportion of accounts')
y_ticks = axes.get_yticks()
axes.set_yticklabels(np.round(y_ticks*50000,2))
axes.legend(['Not defaulted', 'Defaulted'])
axes.set_title('Normalized distributions of credit limits by response variable')
```

A plotagem deve ficar assim:

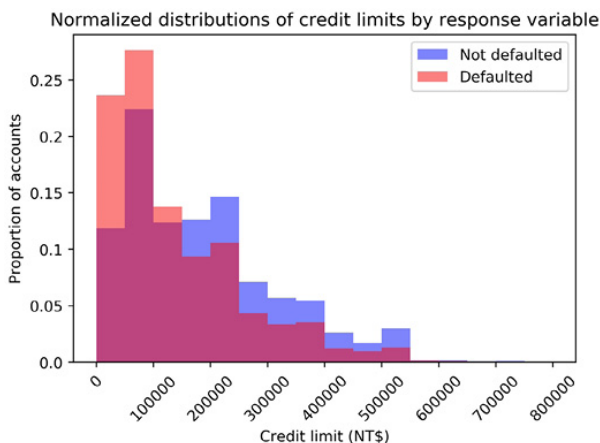


Figura 3.18 – Histogramas duplos normalizados.

Podemos ver que as plotagens do Matplotlib são altamente personalizáveis. Para visualizar todos os diferentes itens que você pode obter (get) e definir (set) nos eixos do Matplotlib, pesquise aqui:

https://matplotlib.org/api/axes_api.html.

O que podemos extrair dessa plotagem? Parece que as contas inadimplentes tendem a ter uma proporção maior de limites de crédito menores. Contas com limites de crédito menores do que NT\$150,000 estão relativamente mais propensas a inadimplir, enquanto o oposto é verdadeiro para contas com limites maiores do que esse. Devemos considerar se isso faz sentido. Nossa hipótese era de que o cliente daria limites menores a contas que apresentassem mais risco. Essa intuição é compatível com as proporções maiores de inadimplentes com limites de crédito menores que observamos aqui.

Dependendo de como ocorrer a construção do modelo, se as características examinadas neste exercício se mostrarem importantes para a modelagem preditiva como esperado, seria bom exibir esses gráficos para nosso cliente, como parte de uma apresentação do trabalho. Isso daria ao cliente uma ideia de como o modelo funciona.

Algo essencial que aprendemos nesta seção é que leva tempo produzir apresentações visuais eficazes. É bom reservar algum tempo no fluxo de trabalho de seu projeto para isso. Apresentações visuais convincentes valem o esforço, já que comunicam descobertas importantes de maneira rápida e eficiente para o cliente. Geralmente elas são uma opção melhor do que adicionar muito texto aos materiais criados. A comunicação visual de conceitos quantitativos é uma habilidade-chave da ciência de dados.

Seleção de características univariada: o que ela pode ou não fazer

Neste capítulo, aprendemos técnicas para percorrer as características uma a uma e ver se elas têm poder preditivo. Essa é uma ótima primeira etapa, e, se você já tem características que sejam boas preditoras da variável de resultado, talvez não precise gastar muito tempo considerando as características antes da modelagem. No entanto, há desvantagens na seleção de características univariada. Especificamente, ela não considera as **interações** entre as características. Por exemplo, e se a taxa de não pagamento de dívidas for muito alta para pessoas com determinado grau de educação e para um intervalo de limite de crédito específico?

Além disso, com os métodos que usamos aqui, só os efeitos lineares das características são capturados. Se uma característica for mais preditiva ao passar por algum tipo de **transformação**, como a

transformação **polinomial** ou **logarítmica**, ou **binning** (**discretização**), as técnicas lineares da seleção de características univariada podem não ser eficazes. As interações e transformações são exemplos de **engenharia de características**, ou criação de novas características, nesses casos a partir de características existentes. As deficiências dos métodos de seleção de características linear podem ser remediadas pelas técnicas de modelagem não lineares que incluem a floresta aleatória, que examinaremos posteriormente. Mesmo assim é importante, e rápido, procurar os relacionamentos simples que podem ser encontrados pelos métodos lineares da seleção de características univariada.

Entendendo a regressão logística com sintaxe de funções Python e a função sigmóide

Nesta seção, abriremos totalmente a caixa preta: obteremos um conhecimento amplo de como a regressão logística funciona. Começaremos introduzindo um novo conceito de programação: as **funções**. Ao mesmo tempo, conheceremos uma função matemática que desempenha um papel-chave na regressão logística.

No sentido mais básico, uma função na programação de computadores é um fragmento de código que recebe entradas e produz saídas. Já usamos funções no decorrer do livro, mas elas foram escritas por outras pessoas. Sempre que você empregar uma sintaxe como esta: `output = do_something_to(input)`, terá usado uma função. Por exemplo, o NumPy tem uma função que você pode usar para calcular a média da entrada:

```
np.mean([1, 2, 3, 4, 5])  
3.0
```

Figura 3.19 – Função de média do NumPy.

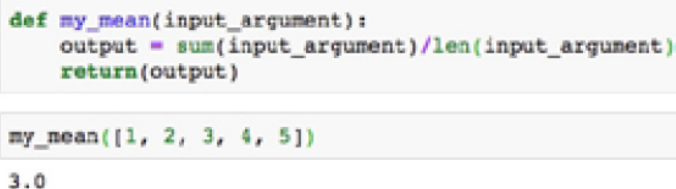
As funções **isolam** as operações que estão sendo executadas, portanto, no nosso exemplo, não precisaremos ver todas as linhas de código necessárias para o cálculo de uma média sempre que tivermos de calculá-la. Já existem versões predefinidas para muitas funções matemáticas comuns em pacotes como o NumPy. Não é preciso “reinventar a roda”. As implementações de pacotes conhecidos são populares por uma razão: pessoas se dedicaram a criá-las da maneira mais eficiente possível. Logo, seria inteligente usá-las. No entanto, já que todos os pacotes que estamos utilizando são **open source**, se você

estiver interessado em ver como as funções das bibliotecas que usamos são implementadas, pode examinar o código existente nelas.

Agora, a título de ilustração, aprenderemos a sintaxe de funções do Python criando nossa própria função para a média aritmética. A sintaxe de funções do Python é semelhante aos blocos `for` ou `if` porque o corpo da função é indentado e a declaração é seguida por dois-pontos. Aqui está o código de uma função que calcula a média:

```
def my_mean(input_argument):  
    output = sum(input_argument)/len(input_argument)  
    return(output)
```

Depois que você executar a célula de código com essa definição, a função ficará disponível em outras células de código do notebook. Por exemplo:



```
def my_mean(input_argument):  
    output = sum(input_argument)/len(input_argument)  
    return(output)  
  
my_mean([1, 2, 3, 4, 5])  
  
3.0
```

Figura 3.20 – Calculando a média com uma função definida pelo usuário.

A primeira parte da definição de uma função, como mostrado aqui, é iniciar uma linha de código com `def`, seguido por um espaço e pelo nome que quisermos dar à função. Depois disso vêm os parênteses, dentro do quais os nomes dos **parâmetros** da função serão especificados. Os parâmetros são os nomes das variáveis de entrada e têm como escopo o corpo da função: os nomes de variáveis definidos como parâmetros ficam disponíveis dentro da função quando ela é **chamada** (usada); eles não ficam disponíveis fora da função. Pode haver mais de um parâmetro; nesse caso, eles são separados por vírgulas. Depois dos parênteses vêm os dois-pontos.

O corpo da função é indentado e pode conter qualquer código que opere com as entradas. Uma vez que essas operações forem concluídas, a última linha deve começar com `return` e conter a(s) variável(is) de saída, separadas por vírgula se houver mais de uma. Estamos deixando de fora muitos pontos importantes nessa introdução muito simples às funções, mas essas são as partes essenciais que você precisa conhecer para começar.

O poder de uma função surge quando a usamos. Observe que, após definirmos a função, podemos **chamá-la** em um bloco de código

separado pelo nome que lhe demos e ela operará com qualquer que sejam as entradas que **passarmos**. É como se copiássemos e colássemos todo o código nesse novo local. Porém, é mais inteligente do que fazer isso. E, se você for usar o mesmo código muitas vezes, uma função pode reduzir bastante o tamanho geral de seu código.

Como uma breve nota adicional, você pode especificar as entradas usando os nomes dos parâmetros explicitamente, o que deve produzir um resultado mais claro quando houver muitas entradas.

```
my_mean(input_argument=[1, 2, 3])  
2.0
```

Figura 3.21 – Usando uma função com um nome de parâmetro.

Agora que estamos familiarizados com os aspectos básicos das funções Python, consideraremos uma função matemática que é importante para a regressão logística, chamada **sigmóide**. Essa função também pode ser chamada de **função logística**. A definição da função sigmóide é:

$$f(X) = \text{sigmoid}(X) = \frac{1}{1 + e^{-X}}$$

Figura 3.22 – Função sigmóide.

Dividiremos as diferentes partes dessa função. Como você pode ver, a função sigmóide envolve o **número irracional e**, que também é conhecido como base do **logaritmo natural**, ao contrário dos logaritmos de base 10 que usamos anteriormente para a exploração de dados. Para calcular e^x usando Python, não precisamos calcular a exponenciação manualmente. O NumPy tem uma função `exp` que usa e como expoente de entrada automaticamente. Se você examinar a documentação, verá que esse processo chama-se usar o “exponencial”, o que soa vago. No entanto, nesse caso fica subentendido que a base do expoente é e . Em geral, quando queremos usar um expoente em Python, como 23 (“dois elevado à terceira potência”), a sintaxe é 2 asteriscos: `2**3`, que é igual a 8, por exemplo.

Consideremos como as entradas podem ser passadas para a função `np.exp`. Já que a implementação do NumPy é **vetorizada**, essa função pode receber tanto números individuais quanto arrays ou matrizes como entrada. Primeiro, calculamos o exponencial de 1, que mostra o valor aproximado de e , e depois e^0 , que é claro é igual a 1, como ocorre com a potência zero de qualquer base:

```
np.exp(1)
2.718281828459045

np.exp(0)
1.0
```

Figura 3.23 – Exp(1) e exp(0) com o NumPy.

Para ilustrar a implementação vetorizada de `np.exp`, criaremos um array de números usando a função `linspace` do NumPy. Essa função recebe como entrada os pontos inicial e final de um intervalo e o número de valores que queremos dele para criar um array com esses valores linearmente espaçados. Ela desempenha um papel semelhante ao da função Python `range`, mas também pode produzir valores decimais:

```
X_exp = np.linspace(-4,4,81)
print(X_exp[:5])
print(X_exp[-5:])

[-4.  -3.9 -3.8 -3.7 -3.6]
[3.6  3.7  3.8  3.9  4. ]

Y_exp = np.exp(X_exp)
plt.plot(X_exp, Y_exp)
plt.title('Plot of  $e^x$ ')
```

Figura 3.24 – Usando `np.linspace` para criar um array.

Já que `np.exp` é vetorizada, calculará eficientemente o exponencial de todo o array de uma só vez. Aqui está o código, com a saída, que calcula o exponencial de nosso array `X_exp` e examina os cinco primeiros valores:

```
Y_exp = np.exp(X_exp)
Y_exp[:5]

array([0.01831564, 0.02024191, 0.02237077, 0.02472353, 0.02732372])
```

Figura 3.25 – Função `exp` do NumPy.

Exercício 13: Plotando a função sigmóide

Neste exercício, usaremos as variáveis `X_exp` e `Y_exp`, criadas anteriormente, para gerar uma plotagem da função exponencial no intervalo `[-4, 4]`. Você precisará de todo o código das *figuras 3.23 e 3.24* para ter essas variáveis disponíveis no exercício. Em seguida, definiremos uma função para o sigmóide, criaremos uma plotagem dela e consideraremos qual sua relação com a função exponencial.

Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2Dz5iNA>.

1. Use este código para plotar a função exponencial:

```
plt.plot(X_exp, Y_exp)  
plt.title('Plot of  $e^X$ ')
```

A plotagem ficará assim:

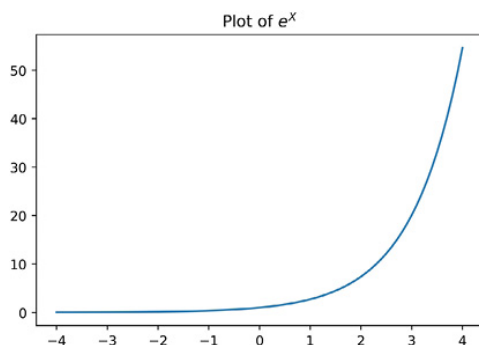


Figura 3.26 – Plotando a função exponencial.

Observe que, ao nomear a plotagem, nos beneficiamos de um tipo de sintaxe chamado **LaTeX**, que permite formatar notação matemática. Não entraremos em detalhes sobre o LaTeX aqui, é suficiente dizer que ele é muito flexível. Repare que inserir parte da string do título entre cifrões faz com que ela seja renderizada com o uso do LaTeX, e que o sobrescrito pode ser criado com o uso de $\wedge$.

Observe também na Figura 3.26 que muitos pontos com pouco espaço entre eles geram uma curva de aparência suave, mas na verdade é um gráfico de pontos distintos conectados por segmentos de linha.

O que podemos aprender com a função exponencial?

Ela nunca é negativa: à medida que X se aproxima do infinito negativo, Y se aproxima de 0.

À medida que X aumenta, no início Y aumenta lentamente, mas com muita rapidez “explode”. É a isso que as pessoas se referem quando usam a expressão “crescimento exponencial” para indicar um aumento rápido.

O que podemos dizer da função sigmóide em relação à função exponencial?

A função sigmóide envolve e^{-x} e não e^x . O gráfico de e^{-x} é apenas o reflexo de e^x sobre o eixo y . Ele pode ser plotado facilmente, com o uso de chaves para gerarmos caracteres em sobrescrito no título da plotagem.

2. Execute este código para ver a plotagem de e^{-x} :

```
Y_exp = np.exp(-X_exp)
plt.plot(X_exp, Y_exp)
plt.title("Plot of  $e^{-X}$ ")
```

A saída deve ter esta aparência:

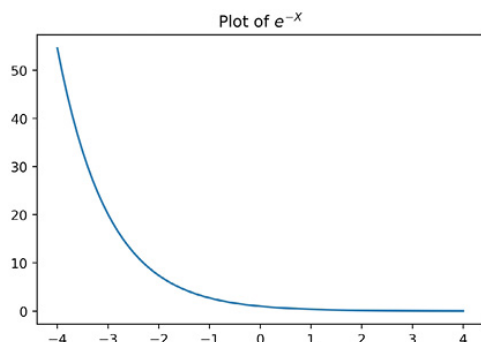


Figura 3.27 – Plotagem de $\exp(-X)$.

Na função sigmóide, e^{-x} fica no denominador, com 1 sendo adicionado. O numerador é 1. Então, o que ocorre na função à medida que X se aproxima do infinito negativo? Sabemos que e^{-x} “aumenta explosivamente”, tornando-se muito alto. O denominador torna-se muito alto e a fração aproxima-se de 0. E no caso em que X aumenta em direção ao infinito positivo? Podemos ver que e^{-x} chega muito perto de 0. Portanto, nesse caso, a função sigmóide seria aproximadamente $1/1 = 1$. Isso nos dá a impressão de que a função fica entre 0 e 1. Implementaremos uma função sigmóide em Python e a usaremos para criar uma plotagem e ver se a realidade coincide com essa impressão.

3. Defina uma função sigmóide como esta:

```
def sigmoid(X):
    Y = 1 / (1 + np.exp(-X))
    return Y
```

4. Crie um intervalo maior de valores de x para usar na plotagem e represente a função sigmóide. Use este código:

```
X_sig = np.linspace(-7,7,141)
Y_sig = sigmoid(X_sig)
plt.plot(X_sig,Y_sig)
```

```
plt.yticks(np.linspace(0,1,11))  
plt.grid()  
plt.title("The sigmoid function")
```

Devemos ver a seguinte plotagem:

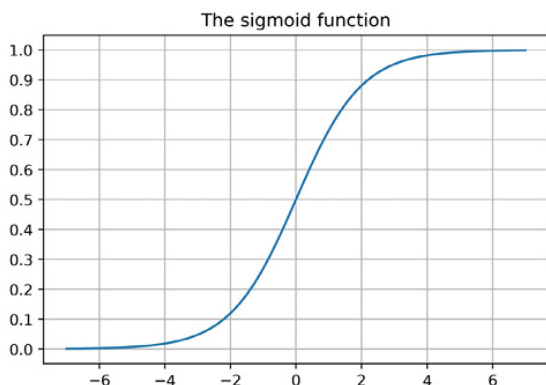


Figura 3.28 – Plotagem da função sigmóide.

Essa imagem coincide com o que esperávamos. Além disso, podemos ver que $\text{sigmoid}(0) = 0.5$. O que a função sigmóide tem de especial? A saída dessa função fica estritamente entre 0 e 1. Essa é uma boa propriedade para uma função que deve prever probabilidades, que também precisam estar entre 0 e 1. Tecnicamente as probabilidades podem ser exatamente iguais a 0 e 1, embora a função sigmóide nunca seja. No entanto, ela chega perto demais para que essa seja uma limitação prática.

Você deve lembrar-se de que descrevemos a regressão logística como produzindo **probabilidades previstas** de associação de classe em vez de prever diretamente a associação. Isso permite uma implementação mais flexível da regressão logística, possibilitando a seleção da probabilidade limite. A função sigmóide é a fonte dessas probabilidades previstas. Em breve veremos como as diferentes características são usadas no cálculo das probabilidades previstas.

Escopo das funções

Quando você começar a usar funções, deve tomar consciência do conceito de **escopo**. Observe que, quando escrevemos a função sigmóide, criamos uma variável, Y , dentro da função. Variáveis criadas dentro de funções são diferentes das criadas fora delas. Elas são criadas e destruídas dentro da própria função, quando ela é chamada. Essas variáveis são consideradas de escopo **local**: elas só funcionam na função. Se você vem executando todo o código como foi

escrito neste capítulo no mesmo notebook e em sequência, repare que não é possível acessar a variável Y após o uso da função sigmóide:

```
Y

-----
NameError                                Traceback (most recent call last)
<ipython-input-46-c881daf1af41> in <module>
----> 1 Y

NameError: name 'Y' is not defined
```

Figura 3.29 – Variável Y fora do escopo do notebook.

A variável Y não faz parte do escopo **global** do notebook. No entanto, variáveis globais criadas fora de funções ficam disponíveis dentro do escopo local destas, mesmo se não forem fornecidas como parâmetros da função. Aqui estamos demonstrando a criação de uma variável com escopo global fora de uma função e o acesso a ela dentro da função. A função não recebe parâmetros, mas, como você pode ver, ela pode usar o valor da variável global para criar uma saída:

```
example_global_variable = 1

def example_function():
    output = example_global_variable + 1
    return(output)

example_function()

2
```

Figura 3.30 – Variável global disponível dentro do escopo local da função.

Nota

Mais detalhes sobre o escopo

O escopo das variáveis pode ser confuso, mas é preciso conhecer o assunto para quando você começar a fazer uso mais avançado das funções. Embora esse conhecimento não seja requerido no livro, é possível obter uma explicação mais aprofundada sobre o escopo das variáveis em Python aqui:

https://nbviewer.jupyter.org/github/rasbt/python_reference/blob/master/tutorials/scope_resolution_legb_rule.ipynb.

Curvas sigmóides em aplicações científicas

Além de serem fundamentais para a regressão logística, as curvas sigmóides são usadas em uma grande variedade de aplicações. Na biologia, elas podem ser usadas para descrever o crescimento de um organismo, que começa lentamente e, então, passa por uma fase mais veloz, seguida de uma suave perda de potência quando o tamanho final é alcançado. A curva sigmóide também pode ser usada para descrever o crescimento

populacional, que tem uma trajetória semelhante, aumentando rapidamente, mas diminuindo o ritmo quando a tolerância do ambiente é alcançada.

Por que a regressão logística é considerada um modelo linear?

Mencionamos anteriormente que a regressão logística é considerada um **modelo linear** enquanto examinávamos se o relacionamento entre as características e a resposta lembrava um relacionamento linear. Você deve se lembrar de que plotamos uma análise groupby/mean para a característica EDUCATION no *Capítulo 1, Exploração e limpeza de dados* assim como para a característica PAY_1 neste capítulo para ver se as taxas de inadimplência entre os valores dessas características exibiam tendência linear. Embora essa seja uma boa maneira de termos uma noção geral de quanto as características “podem ou não ser lineares”, agora formalizaremos o conceito de por que a regressão logística é um modelo linear.

Um modelo é considerado linear quando a transformação usada no cálculo da previsão é uma **combinação linear** das características. As possibilidades de uma combinação linear ocorrem quando cada característica pode ser multiplicada por uma constante numérica, esses termos podem ser somados, e uma constante adicional também pode ser somada. Por exemplo, em um modelo simples com duas características, X_1 e X_2 , uma combinação linear teria esta forma:

$$\text{Combinação linear de } X_1 \text{ e } X_2 = \theta_0 + \theta_1 X_1 + \theta_2 X_2$$

Figura 3.31 – Combinação linear de X_1 e X_2 .

As constantes θ_i podem ser qualquer número, positivo, negativo ou zero, para $i = 0, 1$ e 2 (mas, se o coeficiente for 0, isso removerá a característica da combinação linear). Um exemplo familiar de uma transformação linear com uma única variável seria uma linha reta com a equação $y = mx + b$. Nesse caso, $\theta_0 = b$ e $\theta_1 = m$. θ_0 é chamada de interceptação da combinação linear, o que faz sentido quando pensamos em uma equação de linha reta com essa forma de inclinação-interceptação.

O que “não é permitido” nas transformações lineares? Qualquer expressão matemática diferente da que acabei de descrever, como:

- Multiplicar uma característica por ela própria; por exemplo, X_1^2 ou X_1^3 . Esses termos são chamados de termos polinomiais.

- Multiplicar características agrupadas; por exemplo, X_1X_2 . Isso é chamado de interação.
- Aplicar transformações não lineares às características; por exemplo, logaritmo e raiz quadrada.
- Outras funções matemáticas complexas.
- Equações do tipo “se, então”. Por exemplo, “se $X_1 > a$, então $y = b$ ”.

No entanto, embora essas transformações não façam parte da formulação básica de uma combinação linear, elas podem ser adicionadas a um modelo linear com a **engenharia de características**, por exemplo, com a definição de uma nova característica, $X_3 = X_1^2$.

Aprendemos anteriormente que as previsões da regressão logística, que assumem a forma de probabilidades, são feitas com o uso da função sigmóide. Essa função é claramente não linear e tem a seguinte equação:

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

Figura 3.32 – Função sigmóide não linear.

Por que, então, a regressão logística é considerada um modelo linear? A resposta para essa pergunta está em uma formulação diferente da equação sigmóide, chamada função logit. Podemos derivar a função logit resolvendo a função sigmóide para X ; em outras palavras, encontrando o inverso da função sigmóide. Primeiro, definimos sigmóide igual a p , a probabilidade de observação da classe positiva, e depois damos a resolução para X como mostrado na Figura 3.33:

$$\begin{aligned} p &= \frac{1}{1+e^{-x}} \\ 1+e^{-x} &= \frac{1}{p} \\ e^{-x} &= \frac{1}{p} - 1 \\ e^{-x} &= \frac{1-p}{p} \\ e^x &= \frac{p}{1-p} \\ x &= \log\left(\frac{p}{1-p}\right) \end{aligned}$$

Figura 3.33 – Resolvendo para X .

Aqui, usamos algumas leis dos expoentes e logaritmos para achar a

solução para X . Você também deve ver $\logit$ expressa como:

$$X = \log\left(\frac{p}{q}\right)$$

Figura 3.34 – Função $\logit$.

A **probabilidade de falha**, q , é expressa em termos da **probabilidade de sucesso**, p : $q = 1 - p$, porque a soma das probabilidades é 1. Ainda que em nosso caso o não pagamento de dívidas possa ser considerado uma falha para resultados do mundo real, o resultado positivo (variável de resposta = 1 em um problema binário) é convencionalmente considerado um “sucesso” na terminologia matemática. A função $\logit$ também é chamada de **logaritmo das chances (log-odds)**, porque é o logaritmo natural da **razão de possibilidades (odds ratio)**, p/q . Estamos familiarizados com as razões de possibilidades porque elas são usadas no mundo dos esportes em frases como “as chances são de 2 para 1 de que o time a derrote o time b”.

De forma geral, o que chamamos de X maiúsculo nessas manipulações representa a combinação linear de todas as características. Por exemplo, teríamos $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2$ em nosso caso simples de duas características. A regressão logística é considerada um modelo linear porque as características incluídas em X na verdade só estão sujeitas a uma combinação linear quando a variável de resposta é considerada a razão de possibilidades. Essa é uma maneira alternativa de formulação do problema, em comparação com a equação sigmóide.

Resumindo, as características $X_1, X_2, \dots, X_j$ aparecem da seguinte forma na versão de equação sigmóide da regressão logística:

$$p = \frac{1}{1 + e^{-(\theta_0 + \theta_1 X_1 + \theta_2 X_2 + \dots + \theta_j X_j)}}$$

Figura 3.35 – Versão sigmóide da regressão logística.

No entanto, têm essa aparência na versão de logaritmo das chances, e é por isso que a regressão logística é chamada de modelo linear:

$$\theta_0 + \theta_1 X_1 + \theta_2 X_2 + \dots + \theta_j X_j = \log\left(\frac{p}{q}\right)$$

Figura 3.36 – Versão de logaritmo das chances da regressão logística.

Se considerarmos essa forma de regressão logística, idealmente as características do modelo seriam **lineares no logaritmo das chances**

da variável de resposta. Veremos o que isso significa no próximo exercício.

A regressão logística faz parte de uma classe mais ampla de modelos estatísticos chamados **Modelos Lineares Generalizados (GLMs, Generalized Linear Models)**. Os GLMs estão ligados ao conceito básico de regressão linear ordinária, que pode ter uma única característica (isto é, a **linha de melhor ajuste**, $y = mx + b$, para uma característica individual, x) ou mais de uma na **regressão linear múltipla**. A conexão matemática entre os GLMs e a regressão linear é a **função de ligação**. A função de ligação da regressão logística é a função logit que acabamos de conhecer.

Exercício 14: Examinando a conveniência das características para a regressão logística

No *Exercício 12: Visualizando o relacionamento entre as características e a resposta*, plotamos uma operação `groupby/mean` para a característica que talvez seja a mais importante do modelo, de acordo com nossa exploração até agora: `PAY_1`. Ao agrupar amostras pelos valores de `PAY_1` e verificar a média da variável de resposta, na verdade estamos examinando a probabilidade p de inadimplência dentro de cada um desses grupos.

Neste exercício, avaliaremos a conveniência do uso de `PAY_1` na regressão logística. Faremos isso examinando o logaritmo das chances de inadimplência dentro desses grupos para ver se a variável de resposta é linear, como supõe formalmente a regressão logística. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2Dz5iNA>.

1. Veja se ainda consegue acessar as variáveis do *Exercício 12, Visualizando o relacionamento entre as características e a resposta* no notebook examinando com o código a seguir o DataFrame do valor de média da variável de resposta para diferentes valores de `PAY_1`:

```
group_by_pay_mean_y
```

A saída deve ser:

group_by_pay_mean_y	
default payment next month	
PAY_1	
-2	0.131664
-1	0.170002
0	0.128295
1	0.336400
2	0.694701
3	0.773973
4	0.682540
5	0.434783
6	0.545455
7	0.777778
8	0.588235

Figura 3.37 – Taxas de inadimplência dentro de grupos de valores de PAY_1 como probabilidades de não pagamento.

- Extraia os valores de média da variável de resposta para esses grupos e insira-os em uma variável, *p*, representando a probabilidade de inadimplência:

```
p = group_by_pay_mean_y['default payment next month'].values
```

- Crie uma probabilidade, *q*, de inexistência de inadimplência. Já que há apenas dois resultados possíveis nesse problema binário, e a soma das probabilidades é 1, é muito fácil calcular *q*. Exiba também os valores de *p* e *q* para confirmar:

```
q = 1-p
print(p)
print(q)
```

A saída deve ser:

```
p = group_by_pay_mean_y['default payment next month'].values
q = 1-p
print(p)
print(q)

[0.13166397 0.17000198 0.12829525 0.33639988 0.69470143 0.7739726
 0.68253968 0.43478261 0.54545455 0.77777778 0.58823529]
[0.86833603 0.82999802 0.87170475 0.66360012 0.30529857 0.2260274
 0.31746032 0.56521739 0.45454545 0.22222222 0.41176471]
```

Figura 3.38 – Calculando q.

- Calcule a razão de possibilidades de *p* e *q*, assim como o logaritmo das chances, usando a função de logaritmo natural do NumPy:

```
odds_ratio = p/q
log_odds = np.log(odds_ratio)
log_odds
```

Você deve obter esta saída:

```
odds_ratio = p/q
odds_ratio

array([0.15162791, 0.20482215, 0.14717742, 0.50693161, 2.27548209,
       3.42424242, 2.15        , 0.76923077, 1.2        , 3.5        ,
       1.42857143])

log_odds = np.log(odds_ratio)
log_odds

array([-1.88632574, -1.58561322, -1.91611649, -0.67937918, 0.82219194,
       1.23088026, 0.76546784, -0.26236426, 0.18232156, 1.25276297,
       0.35667494])
```

Figura 3.39 – Razão de possibilidades e logaritmo das chances.

5. Para plotar o logaritmo das chances para os valores da característica, podemos obter os valores no índice do DataFrame que contém groupby/mean. Você pode exibir o índice desta forma:

```
group_by_pay_mean_y.index
```

Essa linha deve produzir a saída a seguir:

```
group_by_pay_mean_y.index
Int64Index([-2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8], dtype='int64', name='PAY_1')
```

Figura 3.40 – Como obter valores a partir do índice de um DataFrame do pandas.

6. Crie uma plotagem semelhante à que criamos anteriormente, para exibir o logaritmo das chances dos valores da característica. Aqui está o código:

```
plt.plot(group_by_pay_mean_y.index, log_odds, '-x')
plt.ylabel('Log odds of default')
plt.xlabel('Values of PAY_1')
```

A plotagem deve ficar assim:

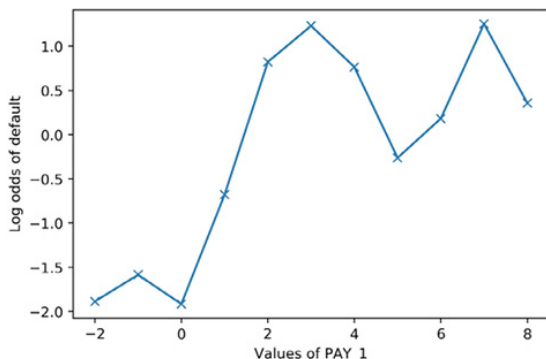


Figura 3.41 – Logaritmo das chances de inadimplência para valores de PAY_1.

Podemos ver nessa plotagem que o relacionamento entre o logaritmo das chances da variável de resposta e a característica PAY_1 não é tão diferente do relacionamento entre a taxa de inadimplência e essa característica, que plotamos no *Exercício 12, Visualizando o relacionamento entre as características e a resposta*. Logo, se a “taxa de inadimplência” for um conceito mais simples para você comunicar para o sócio da empresa, pode ser melhor usá-lo. No entanto, em termos de entendimento do funcionamento da regressão logística, essa plotagem mostra exatamente o que se supõe que seja linear.

O ajuste de linha reta é um bom modelo para esses dados?

Parece que uma “linha de melhor ajuste” desenhada nessa plotagem subiria da esquerda para a direita. Ao mesmo tempo, os dados não dão a impressão de que resultariam de um processo realmente linear. Uma maneira de olharmos os dados seria considerar que os valores -2, -1 e 0 parecem se enquadrar em um regime de logaritmo das chances diferente dos outros. PAY_1 = 1 é intermediário e o resto é em grande parte mais alto. Talvez uma engenharia de características baseada nessa variável, ou diferentes maneiras de codificar as categorias representadas por -2, -1 e 0 seriam mais eficazes para a modelagem. Lembre-se disso enquanto continuamos a modelar os dados com uma regressão logística e depois usando outras abordagens.

Dos coeficientes da regressão logística às previsões com o uso da função sigmóide

Antes do próximo exercício, examinaremos como os coeficientes de uma regressão logística são usados para calcular probabilidades previstas e depois faremos previsões para a classe da variável de resposta.

Lembre-se de que a regressão logística prevê a probabilidade de associação de classe, de acordo com a equação sigmóide. No caso de duas características com uma interceptação, a equação é:

$$p = \frac{1}{1 + e^{-(\theta_0 + \theta_1 X_1 + \theta_2 X_2)}}$$

Figura 3.42 – Função sigmóide para prever a probabilidade de associação de classe para duas características.

Quando chamamos o método `.fit` em um objeto de modelo de regressão logística no scikit-learn usando os dados de treinamento,

como demonstramos várias vezes, os parâmetros (a interceptação e os coeficientes) θ_0 , θ_1 e θ_2 são estimados a partir dos dados de treinamento rotulados. Na verdade, o scikit-learn descobre como selecionar valores para θ_0 , θ_1 e θ_2 de modo a classificar corretamente o maior número de pontos de dados de treinamento. Conheceremos melhor como esse processo funciona no próximo capítulo.

Quando chamamos `.predict`, o scikit-learn calcula as probabilidades previstas de acordo com os valores de parâmetros ajustados e a equação sigmóide. Uma amostra específica será então classificada como positiva se $p \geq 0.5$; caso contrário, será negativa.

Sabemos que a plotagem da equação sigmóide tem a aparência a seguir (Figura 3.43), em que o eixo x mostra a combinação linear das características $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2$.

Observe que, se $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2 \geq 0$ no eixo x , a probabilidade prevista seria $p \geq 0.5$ no eixo y e classificariamos a amostra como positiva. Caso contrário, teríamos $p < 0.5$ e a amostra seria classificada como negativa. Podemos usar essa observação para calcular uma condição linear para a previsão positiva, no que diz respeito às características X_1 e X_2 , usando os coeficientes e a interceptação. Resolvendo a desigualdade da previsão positiva $X = \theta_0 + \theta_1 X_1 + \theta_2 X_2 \geq 0$ para X_2 , podemos obter uma desigualdade linear semelhante à equação linear $y = mx + b$: $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$.

Isso nos ajudará a ver o limite de decisão linear da regressão logística no **espaço de características** X_1 - X_2 no próximo exercício.

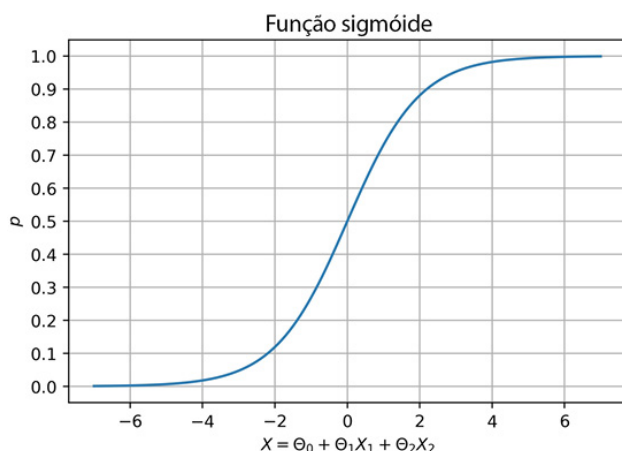


Figura 3.43 – Previsões e classes reais plotadas juntas.

Sabemos agora, do ponto de vista teórico e matemático, por que a

regressão logística é considerada um modelo linear. Além disso, examinamos uma característica individual e consideramos se a suposição de linearidade era apropriada. Também é importante entender a suposição de linearidade no que diz respeito ao que podemos esperar da regressão logística em termos de flexibilidade e poder quando usada como classificador com várias características.

Exercício 15: Limite de decisão linear da regressão logística

Neste exercício, ilustraremos o conceito de **limite de decisão** para uma classificação binária. Usaremos dados sintéticos para criar um exemplo claro de visualização do limite de decisão da regressão logística em relação às amostras de treinamento. Começaremos gerando aleatoriamente duas características, X_1 e X_2 . Já que há duas características, podemos dizer que os dados desse problema são bidimensionais. Isso facilita a visualização. Os conceitos que ilustraremos aqui podem ser generalizados para casos de mais de duas características, como os datasets do mundo real que provavelmente você verá em seu trabalho; no entanto, o limite de decisão é mais difícil de visualizar em espaços com mais dimensões.

Execute as etapas a seguir para fazer o exercício:

Nota: O código e saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2VnbXRL>.

1. Gere as características usando o código a seguir:

```
np.random.seed(seed = 6)
X_1_pos = np.random.uniform(low = 1, high = 7, size = (20,1))
print(X_1_pos[0:3])
X_1_neg = np.random.uniform(low = 3, high = 10, size = (20,1))
print(X_1_neg[0:3])
X_2_pos = np.random.uniform(low = 1, high = 7, size = (20,1))
print(X_2_pos[0:3])
X_2_neg = np.random.uniform(low = 3, high = 10, size = (20,1))
print(X_2_neg[0:3])
```

Não é preciso se preocupar com por que selecionamos esses valores; a plotagem criada posteriormente deixará isso claro. Observe, no entanto, que ao mesmo tempo também atribuiremos a classe real. Como resultado, teremos 20 amostras tanto na classe positiva quanto na negativa, perfazendo um total de 40 amostras, e teremos duas características para cada amostra. Exibiremos os três

primeiros valores de cada característica tanto para a classe positiva quanto para a negativa.

A saída deve ser a seguinte:

```
np.random.seed(seed=6)
X_1_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_1_pos[0:3])
X_1_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_1_neg[0:3])
X_2_pos = np.random.uniform(low=1, high=7, size=(20,1))
print(X_2_pos[0:3])
X_2_neg = np.random.uniform(low=3, high=10, size=(20,1))
print(X_2_neg[0:3])
```



```
[[6.35716091]
 [2.99187883]
 [5.92737474]]
[[3.38132155]
 [8.03046066]
 [8.61519394]]
[[6.35716091]
 [2.99187883]
 [5.92737474]]
[[3.38132155]
 [8.03046066]
 [8.61519394]]
```

Figura 3.44 – Gerando dados sintéticos para um problema de classificação binária.

2. Plote esses dados, colorindo as amostras positivas com vermelho e as negativas com azul. O código da plotagem é:

```
plt.scatter(X_1_pos, X_2_pos, color='red', marker='x')
plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.legend(['Positive class', 'Negative class'])
```

O resultado será:

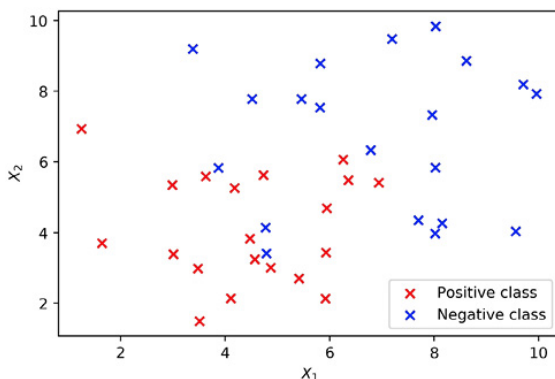


Figura 3.45 – Gerando dados sintéticos para um problema de classificação binária.

Para usar nossas características sintéticas com o scikit-learn, temos

de reuni-las em uma matriz. Usaremos a função `block` do NumPy para fazer isso e criar uma matriz de 40 por 2. Haverá 40 linhas porque há um total de 40 amostras e teremos 2 colunas porque há 2 características. Organizaremos as coisas de modo que as características das amostras positivas estejam nas 20 primeiras linhas e as das amostras negativas venham depois.

3. Crie uma matriz 40 por 2 e exiba a dimensão e as 3 primeiras linhas:

```
X = np.block([[X_1_pos, X_2_pos], [X_1_neg, X_2_neg]])
print(X.shape)
print(X[0:3])
```

A saída deve ser:

```
(40, 2)
[[6.35716091 5.4790643 ]
 [2.99187883 5.3444234 ]
 [5.92737474 3.43664678]]
```

Figura 3.46 – Combinando as características sintéticas em uma matriz.

Também precisamos de uma variável de resposta para acompanhar as características. Sabemos como defini-las, mas precisamos de um array de valores `y` para informar ao scikit-learn.

4. Crie uma pilha vertical (`vstack`) de 20 1's e 20 0's que corresponda à organização das características e redimensione-a para a maneira que o scikit-learn espera. Aqui está o código:

```
y = np.vstack((np.ones((20,1)), np.zeros((20,1)))).reshape(40,)
print(y[0:5])
print(y[-5:])
```

Você obterá a saída a seguir:

```
[1. 1. 1. 1. 1.]
[0. 0. 0. 0. 0.]
```

Figura 3.47 – Crie a variável de resposta para os dados sintéticos.

Estamos prontos para ajustar um modelo de regressão logística para esses dados com o scikit-learn. Usaremos todos os dados como dados de treinamento e examinaremos com que eficácia um modelo linear conseguirá ajustá-los. As próximas etapas devem ser familiares devido ao trabalho em capítulos anteriores sobre como instanciar uma classe de modelo e como ajustar o modelo.

5. Primeiro, importe a classe de modelo usando o código a seguir:

```
from sklearn.linear_model import LogisticRegression
```

6. Agora instancie, indicando o solver `liblinear`, e exiba o objeto de

modelo usando este código:

```
example_lr = LogisticRegression(solver='liblinear')
```

```
example_lr
```

A saída deve ser:

```
from sklearn.linear_model import LogisticRegression
```

```
example_lr = LogisticRegression(solver='liblinear')
```

```
example_lr
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

```
example_lr.fit(X, y)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

Figura 3.48 – Ajuste um modelo de regressão logística para os dados sintéticos no scikit-learn.

7. Treine o modelo com os dados sintéticos:

```
example_lr.fit(X, y)
```

Como ficarão as previsões de nosso modelo ajustado?

Primeiro temos de obter as previsões usando o método `.predict` do modelo treinado, com as mesmas amostras empregadas no treinamento. Em seguida, para adicionar as previsões à plotagem usando o esquema de cores vermelha = classe positiva e azul = classe negativa, criaremos duas listas de índices para utilizar com os arrays, dependendo de se as previsões forem 1 ou 0. Veja se consegue entender a maneira como usamos uma list comprehension incluindo uma instrução `if` para fazê-lo.

8. Use esse código para obter as previsões e separe-as em índices de previsões de classe positiva e negativa. Exiba os índices de previsões de classe positiva como uma verificação:

```
y_pred = example_lr.predict(X)
```

```
positive_indices = [counter for counter in range(len(y_pred)) if y_pred[counter] == 1]
```

```
negative_indices = [counter for counter in range(len(y_pred)) if y_pred[counter] == 0]
```

```
positive_indices
```

A saída deve ser:


```
y_pred = example_lr.predict(X)
```

```
positive_indices = [counter for counter in range(len(y_pred)) if y_pred[counter]==1]  
negative_indices = [counter for counter in range(len(y_pred)) if y_pred[counter]==0]
```

```
positive_indices
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 32, 38]
```

Figura 3.49 – Índices de previsões de classe positiva.

Pelos índices das previsões positivas já podemos dizer que algumas amostras dos dados de treinamento foram classificadas incorretamente: as amostras positivas são as 20 primeiras, mas aqui há índices fora desse intervalo. Você deve ter percebido que um limite de decisão linear não conseguiria classificar corretamente esses dados, conforme nossa discussão. Agora inseriremos as previsões na plotagem, na forma de círculos ao redor de cada ponto de dados e usando cores de acordo com a previsão. Você pode comparar a cor dos símbolos X, os rótulos reais dos dados, com a cor dos círculos (previsões), para ver que pontos foram classificados corretamente e incorretamente.

9. Aqui está o código da plotagem:

```
plt.scatter(X_1_pos, X_2_pos, color='red', marker='x')  
plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x')  
plt.scatter(X[positive_indices,0], X[positive_indices,1], s=150, marker='o',  
            edgecolors='red', facecolors='none')  
plt.scatter(X[negative_indices,0], X[negative_indices,1], s=150, marker='o',  
            edgecolors='blue', facecolors='none')  
plt.xlabel('$X_1$')  
plt.ylabel('$X_2$')  
plt.legend(['Positive class', 'Negative class', 'Positive predictions', 'Negative predictions'])
```

A plotagem deve ficar assim:

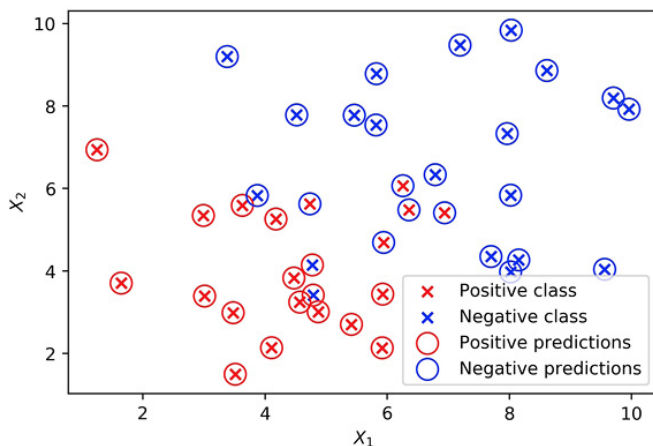


Figura 3.50 – Previsões e classes reais plotadas juntas.

Pela plotagem, fica claro que o classificador tem problemas com os pontos de dados que estão perto de onde seria o limite de decisão linear; alguns deles podem acabar no lado errado desse limite. Como poderíamos definir, e visualizar, a localização real do limite de decisão? De acordo com a seção anterior, sabemos que podemos obter o limite de decisão de uma regressão logística, no espaço de características bidimensional, usando a desigualdade $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$. Já que ajustamos o modelo, podemos recuperar os coeficientes θ_1 e θ_2 , assim como a interceptação θ_0 , para trazê-los para essa equação e criar a plotagem.

10. Use este código para obter os coeficientes do modelo ajustado e exibi-los:

```
theta_1 = example_lr.coef_[0][0]
theta_2 = example_lr.coef_[0][1]
print(theta_1, theta_2)
```

A saída deve ser esta:

-0.20245058016285858 -0.253364236267732

Figura 3.51 – Coeficientes do modelo ajustado.

11. Use o código a seguir para obter a interceptação:

```
theta_0 = example_lr.intercept_
```

Agora use os coeficientes e a interceptação para definir o limite de decisão linear. Isso capturará a linha divisória da desigualdade $X_2 \geq -(\theta_1/\theta_2)X_1 - (\theta_0/\theta_2)$:

```
X_1_decision_boundary = np.array([0, 10])
X_2_decision_boundary = -(theta_1/theta_2)*X_1_decision_boundary - (theta_0/theta_2)
```

Resumindo as últimas etapas, após usar os métodos `.coef_` e `.intercept_` para recuperar os coeficientes θ_1 , θ_2 e a interceptação θ_0 do modelo, fizemos uso deles para criar uma linha definida por dois-pontos, de acordo com a equação que descrevemos para o limite de decisão.

12. Plote o limite de decisão usando o código a seguir, com alguns ajustes para atribuir os rótulos corretos para a legenda e movê-la para um local (loc) fora da plotagem se ela estiver ficando abarrotada:

```
pos_true = plt.scatter(X_1_pos, X_2_pos, color='red', marker='x',
    label='Positive class')
neg_true = plt.scatter(X_1_neg, X_2_neg, color='blue', marker='x',
```

```

label='Negative class')
pos_pred = plt.scatter(X[positive_indices,0], X[positive_indices,1], s=150,
    marker='o', edgecolors='red', facecolors='none', label='Positive predictions')
neg_pred = plt.scatter(X[negative_indices,0], X[negative_indices,1], s=150,
    marker='o', edgecolors='blue', facecolors='none', label='Negative predictions')
dec = plt.plot(X_1_decision_boundary, X_2_decision_boundary, 'k-', label='Decision
    boundary')
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.legend(loc=[0.25, 1.05])

```

Você obterá a seguinte plotagem:

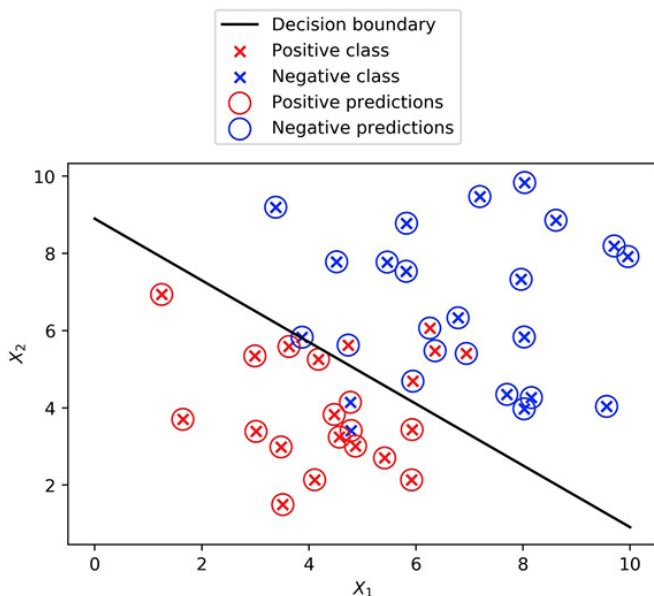


Figura 3.52 – Classes reais, classes previstas e o limite de decisão de uma regressão logística.

Qual é a diferença no limite de decisão em comparação com onde você achou que ele estaria?

Consegue perceber que um limite de decisão linear nunca classificaria com perfeição esses dados?

Como uma maneira de resolver esse problema, poderíamos criar **características obtidas por engenharia** a partir de características existentes, como polinômios e interações, para permitir limites de decisão mais complexos e não lineares na regressão logística. Ou poderíamos usar modelos não lineares como a floresta aleatória, que também pode fazer isso, como veremos posteriormente.

Como observação final, esse exemplo foi visualizado facilmente em

duas dimensões porque há apenas duas características. Em geral, o limite de decisão pode ser descrito por um **hiperplano**, que é a generalização de uma linha reta para espaços multidimensionais. No entanto, a natureza restritiva do limite de decisão linear também é um problema nos hiperplanos.

Atividade 3: Ajustando um modelo de regressão logística e usando os coeficientes diretamente

Nesta atividade, treinaremos um modelo de regressão logística com as duas características mais importantes que descobrimos na exploração de características univariada e aprenderemos a implementar a regressão logística manualmente usando coeficientes do modelo ajustado. Isso mostrará como podemos utilizar a regressão logística em um ambiente de computação em que o scikit-learn não esteja disponível, mas as funções matemáticas necessárias para o cálculo da função sigmóide estejam. Após a conclusão bem-sucedida da atividade, você notará que o valor de ROC AUC calculado com previsões do scikit-learn e o obtido de previsões manuais são iguais: aproximadamente 0.63.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e a saída resultante desta atividade foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2Dz5iNA>.

1. Crie uma divisão treinamento/teste (80/20) com `PAY_1` e `LIMIT_BAL` como características.
2. Importe `LogisticRegression` com as opções padrão, mas configure o solver com `'liblinear'`.
3. Treine o modelo com os dados de treinamento e obtenha as classes previstas, assim como as probabilidades das classes, usando os dados de teste.
4. Extraia os coeficientes e a interceptação do modelo treinado e calcule manualmente as probabilidades previstas. Você terá de adicionar uma coluna de 1's às suas características para multiplicar pela interceptação.
5. Usando um limite de 0.5, calcule manualmente as classes previstas. Compare o resultado com a saída de previsões de classes do scikit-learn.

6. Calcule o ROC AUC usando as probabilidades previstas pelo scikit-learn e as que você calculou manualmente e compare.

Nota: A solução desta atividade pode ser encontrada na página XXX.

Resumo

Neste capítulo, aprendemos como explorar as características individualmente, usando métodos de seleção de características univariada, inclusive a correlação de Pearson e um teste F Anova. Embora examinar as características dessa forma nem sempre forneça um quadro geral, já que podemos estar perdendo interações importantes entre as características, trata-se de uma etapa necessária. Entender os relacionamentos entre as características mais preditivas e a variável de resposta, e criar visualizações eficazes a partir deles, é uma ótima maneira de comunicar as descobertas para o cliente. Também usamos plotagens personalizadas, como a sobreposição de histogramas criada pelo Matplotlib, para gerar visualizações das características mais importantes.

Em seguida, começamos uma descrição detalhada de como a regressão logística funciona, examinando tópicos como a função sigmóide, o logaritmo das chances e o limite de decisão linear. Embora a regressão logística constitua um dos modelos de classificação mais simples e com frequência não seja tão poderosa quanto os outros métodos, é um dos mais usados na classificação e é a base de modelos mais sofisticados como as redes neurais profundas. Logo, um conhecimento detalhado da regressão logística pode ser útil quando você examinar tópicos mais avançados em machine learning. E, em alguns casos, uma simples regressão logística pode ser suficiente. À parte outras considerações que devem ser levadas em conta, o modelo mais simples que atenda aos requisitos costuma ser o melhor modelo.

Se você dominar os materiais deste e do próximo capítulo, estará preparado para usar a regressão logística em seu trabalho. No próximo capítulo, nos basearemos nos fundamentos que aprendemos aqui para ver como os coeficientes são estimados para uma regressão logística, e como ela pode ser usada eficientemente com um número maior de características e também para a seleção destas.

O trade-off entre viés e variância

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Descrever a função custo de perda logarítmica da regressão logística.
- Implementar o procedimento de gradiente descendente para estimar parâmetros do modelo.
- Articular as suposições estatísticas formais do modelo de regressão logística.
- Caracterizar o trade-off entre viés e variância e usá-lo para melhorar os modelos.
- Formular o lasso e a regularização ridge e usá-los no scikit-learn.
- Projetar uma função para selecionar hiperparâmetros de regularização por validação cruzada.
- Criar características de interação por engenharia para melhorar um modelo de subajuste.

Este capítulo apresentará os últimos detalhes da regressão logística e o equipará com as ferramentas para a melhoria do underfitting e do overfitting empregando a regularização e a simples engenharia de características.

Introdução

Neste capítulo, introduziremos os detalhes finais da regressão logística. Além de ser capaz de usar o scikit-learn para ajustar modelos de regressão logística, você aprenderá o procedimento de gradiente descendente, que é semelhante aos processos usados “em segundo plano” para o ajuste de modelos. Para concluir, terminaremos nossa discussão do modelo básico de regressão logística nos familiarizando

com as suposições estatísticas formais desse método.

Começaremos nossa exploração dos conceitos básicos de overfitting, underfitting e trade-off entre viés e variância do machine learning examinando como o modelo de regressão logística pode ser estendido para resolver o problema de overfitting. Após verificar os detalhes matemáticos dos métodos de regularização que são usados para minimizar o overfitting, você aprenderá uma prática útil para ajustar os hiperparâmetros de regularização: a validação cruzada. Por meio dos métodos de regularização e da engenharia de características, veremos como melhorar os modelos tanto de sobreajuste (overfit) quanto de subajuste (underfit).

Estimando os coeficientes e as intercepções da regressão logística

No capítulo anterior, aprendemos que os coeficientes (cada um acompanha uma característica específica) e a intercepção são determinados quando o método `.fit` é chamado em um modelo de regressão logística no scikit-learn usando os dados de treinamento. Esses números são denominados **parâmetros** do modelo e o processo de descobrir os melhores valores para eles é chamado de **estimativa** de parâmetros. Uma vez que os parâmetros são encontrados, o modelo de regressão logística é basicamente um produto acabado; logo, com esses números podemos usar a regressão logística treinada em qualquer ambiente em que seja possível executar funções matemáticas comuns.

É claro que o processo de estimativa de parâmetros é importante, já que nos permite criar um modelo funcional usando nossos dados. No entanto, como a estimativa de parâmetros funciona? Para entender isso, a primeira etapa é nos familiarizarmos com o conceito de **função custo**. Uma função custo é uma maneira de sabermos o quanto as previsões do modelo estão distantes de descrever perfeitamente os dados; ou seja, quanto maiores os erros entre as previsões do modelo e os dados reais, maior o “custo” retornado pela função custo. Esse é um conceito simples para problemas de regressão: a diferença entre as previsões e os valores reais pode ser usada para mostrar o custo, após passar por uma transformação (como a de valor absoluto ou quadratura) que torna o valor do custo positivo e, em seguida, com o cálculo da média de todas as amostras de treinamento.

Para problemas de classificação, principalmente no ajuste de modelos de regressão logística, uma função custo típica é a de perda logarítmica (**log-loss**), também chamada de perda de entropia cruzada. Essa é a função custo que o scikit-learn usa, em uma forma modificada, para ajustar a regressão logística. Aqui está a definição da função de perda logarítmica:

$$\text{Perda logarítmica} = \frac{1}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figura 4.1 – Função de perda logarítmica.

Aqui, há n amostras de treinamento, y_i é o rótulo real (0 ou 1) da i ésima amostra, p_i é a probabilidade prevista de que o rótulo da i ésima amostra é igual a 1 e $\log$ é o logaritmo natural. Uma notação de somatório (isto é, a letra grega maiúscula Sigma) englobando todas as amostras de treinamento e a divisão por n servem para indicar o cálculo de média dessa função custo com todas as amostras de treinamento. Com isso em mente, examine o gráfico a seguir da função de logaritmo natural e considere qual é a interpretação dessa função custo:

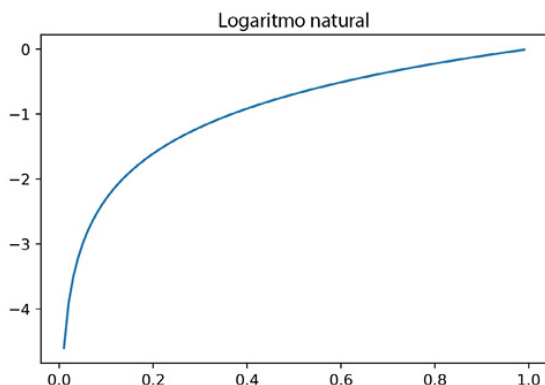


Figura 4.2 – Logaritmo natural no intervalo (0, 1).

Para ver como a função custo de perda logarítmica funciona, considere seu valor para uma amostra positiva; $y = 1$ para uma amostra positiva, logo, essa parte da função custo – isto é, $(1 - y_i) \log(1 - p_i)$ – será exatamente igual a 0 e não afetará o valor. Nesse caso, o valor da função custo é $-y_i \log(p_i) = -\log(p_i)$, já que $y_i = 1$. Portanto, o custo dessa amostra é simplesmente o negativo do logaritmo natural da probabilidade prevista. Como a amostra é na verdade positiva, como a função custo deve se comportar? Esperamos que, para as probabilidades previstas que estejam próximas de 1, a função custo seja pequena, representando um pequeno erro para previsões que

estiverem próximas do valor real. Para previsões que estiverem próximas de 0, ela será maior, já que a função custo deve assumir valores maiores quanto maior for o “erro” na previsão.

Pelo gráfico do logaritmo natural na *Figura 4.1*, podemos ver que, para valores de p que estejam mais próximos de 0, o logaritmo natural resulta em valores negativos cada vez maiores. Isso significa que a função custo resultará em valores positivos cada vez maiores para que o custo de classificar uma amostra positiva com probabilidade muito baixa seja relativamente alto, como esperado. No entanto, se a probabilidade prevista estiver próxima de 1, o gráfico indica que o custo ficará mais próximo de 0 – novamente é isso que é esperado para uma previsão “mais correta”. Logo, a função custo se comporta como esperado para uma amostra positiva. Uma observação semelhante pode ser obtida para amostras negativas.

Agora sabemos como a função custo de perda logarítmica funciona para a regressão logística. No entanto, o que isso tem a ver com como os coeficientes e a interceptação são determinados? Esse será o assunto da próxima seção.

Gradiente descendente para a descoberta de valores de parâmetros ótimos

O problema de encontrar os valores dos parâmetros (coeficientes e interceptação) de um modelo de regressão logística usando um custo de perda logarítmica se resume a uma questão de **otimização**: gostaríamos de encontrar o conjunto de parâmetros que resultasse no custo **mínimo**, já que o menor custo teórico possível é 0 e os custos são maiores para previsões piores. Em outras palavras, queremos o conjunto de parâmetros que seja o “menos errado” na média para todas as amostras de treinamento. Esse processo é executado automaticamente pelo método `.fit` do modelo de regressão logística no `scikit-learn`. Há diferentes técnicas para a descoberta do conjunto de parâmetros de menor custo e você pode escolher qual gostaria de usar com a palavra-chave `solver` quando estiver instanciando a classe do modelo. Esses métodos funcionam de forma um pouco diferente. No entanto, são todos baseados no conceito de **gradiente descendente**.

Nossa tarefa é encontrar o melhor conjunto de parâmetros (isto é, os coeficientes e a interceptação). Os “melhores” parâmetros são os que reduzem a função custo. Na prática, esse processo começa com um **palpite inicial**. A escolha do palpite inicial não é tão importante para

a regressão logística e você não precisa fornecê-lo manualmente; isso é manipulado pela palavra-chave `solver`. Porém, para algoritmos de machine learning mais avançados como as redes neurais profundas, a seleção dos palpites iniciais para parâmetros requer mais atenção.

A título de ilustração, consideraremos um problema em que há apenas um parâmetro a ser estimado. Examinaremos o valor de uma função custo hipotética ($y = f(x) = x^2 - 2x$) e projetaremos um procedimento de gradiente descendente para encontrar o valor do parâmetro, x , para o qual o custo, y , seja o menor. Seleccionaremos alguns valores para x , criaremos uma função que retorne o valor da função custo, e examinaremos esse valor para o intervalo de parâmetros.

O código que faz isso é o seguinte:

```
X_poly = np.linspace(-3,5,81)
print(X_poly[:5], '...', X_poly[-5:])
```

Aqui está a saída da instrução `print`:

```
[-3. -2.9 -2.8 -2.7 -2.6] ... [4.6 4.7 4.8 4.9 5.]
```

O fragmento de código restante é:

```
def cost_function(X):
    return X * (X-2)
y_poly = cost_function(X_poly)
plt.plot(X_poly, y_poly)
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.title('Error surface')
```

A plotagem resultante deve ser:

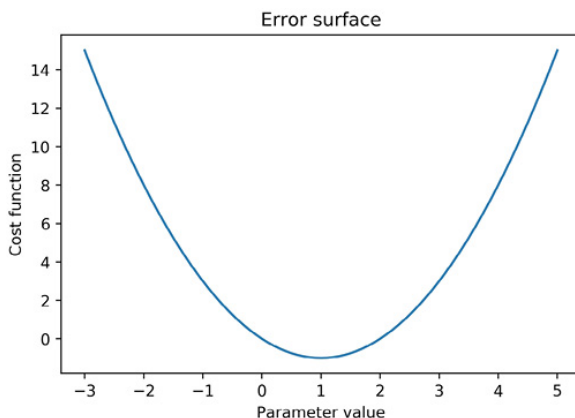


Figura 4.3 – Plotagem da função custo.

Se examinarmos a **superfície de erro** da Figura 4.3, que é a plotagem

da função custo no decorrer de um intervalo de valores de parâmetro, fica claro que o valor do parâmetro resultará no menor valor da função custo: $x = 1$. Na verdade, com uma integral você pode calcular facilmente o valor mínimo definindo a derivada igual a zero e resolvendo x , para confirmar que $x = 1$ é o mínimo. Contudo, nem sempre é possível resolver o problema tão facilmente. Em casos em que é necessário usar um gradiente descendente, nem sempre temos conhecimento da aparência da superfície de erro inteira. Em vez disso, após selecionar o palpite inicial para o parâmetro, tudo que conseguimos saber é a direção da superfície de erro na vizinhança imediata desse ponto.

O **gradiente descendente** é um algoritmo iterativo; partindo do palpite inicial, tentamos encontrar um novo palpite que reduza a função custo e continuamos fazendo isso até encontrar uma boa solução. Tentamos “descer” na superfície de erro, mas só sabemos em que direção nos mover e até onde ir, seguindo a forma da superfície de erro na vizinhança imediata de nosso palpite atual. Em termos matemáticos, só sabemos a **derivada** (que é chamada de **gradiente** em mais de uma dimensão) no valor do parâmetro do palpite atual. Se você não estudou cálculo integral, considere o gradiente como o recurso que diz que direção indica descida e o quanto esta é íngreme a partir de onde estamos. Usamos essas informações para “dar um passo” na direção da diminuição do erro. O tamanho do passo vai depender da **taxa de aprendizado**. Já que o gradiente desce na direção da diminuição do erro, queremos dar um passo na direção que represente o seu negativo.

Essas noções podem ser formalizadas na equação a seguir para chegarmos ao novo palpite, x_{novo} , a partir do palpite atual, x_{anterior} , em que $f'(x_{\text{anterior}})$ é a derivada (isto é, o gradiente) da função custo no palpite atual:

$$x_{\text{novo}} = x_{\text{anterior}} - f'(x_{\text{anterior}}) \times \text{taxa de aprendizado}$$

Figura 4.4 – Equação para obtenção do novo palpite a partir do palpite atual.

Na Figura 4.5, podemos ver os resultados do começo de um procedimento de gradiente descendente a partir de $x = 4.5$, com uma taxa de aprendizado de 0.75, e a otimização de x para a obtenção do menor valor da função custo.

O gradiente descendente também funciona em espaços com mais

dimensões; em outras palavras, com mais de um parâmetro. No entanto, só podemos visualizar uma superfície de erro com até duas dimensões (isto é, dois parâmetros ao mesmo tempo em uma plotagem tridimensional) no mesmo gráfico.

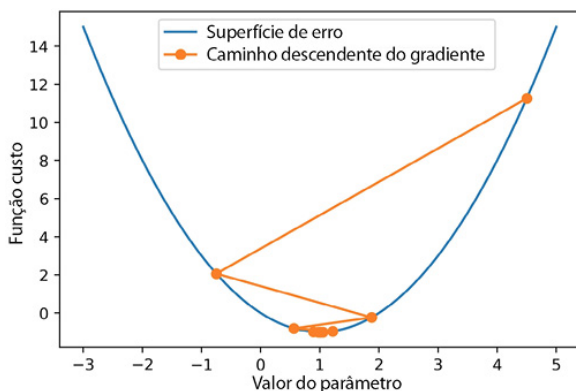


Figura 4.5 – Caminho descendente do gradiente.

Agora que descrevemos o funcionamento do gradiente descendente, faremos um exercício para implementar o algoritmo, baseando-nos no exemplo desta seção.

Exercício 16: Usando o gradiente descendente para reduzir a função custo

Neste exercício, nossa missão é encontrar o melhor conjunto de parâmetros para reduzir a seguinte função custo hipotética: $y = f(x) = x^2 - 2x$. Para fazê-lo, empregaremos o gradiente descendente, que foi descrito na seção anterior. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante dos Exercícios 16 a 18 e da Atividade 4 foram carregados em um Jupyter Notebook que pode ser encontrado em <http://bit.ly/2ZAy2Pr>. Você pode rolar para a seção apropriada dentro do Jupyter Notebook para localizar o exercício ou a atividade que deseja.

1. Crie uma função que retorne o valor da função custo e examine-o ao longo de um intervalo de parâmetros. Você pode usar o código a seguir para fazer isso (observe que estamos repetindo o código da seção anterior):

```
X_poly = np.linspace(-3,5,81)
print(X_poly[:5], '...', X_poly[-5:])
def cost_function(X):
```

```

return X * (X-2)
y_poly = cost_function(X_poly)
plt.plot(X_poly, y_poly)
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.title('Error surface')

```

Você obterá a seguinte plotagem da função custo:

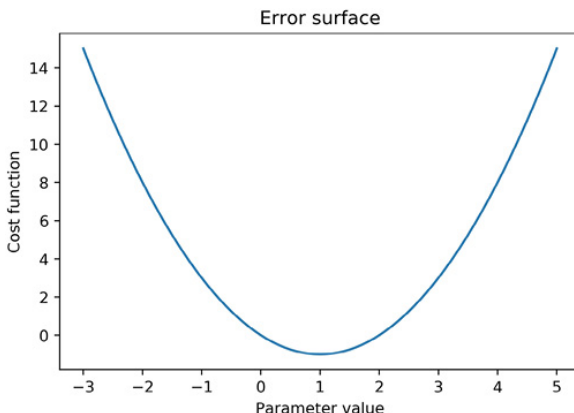


Figura 4.6 – Plotagem da função custo.

2. Crie uma função para o valor do gradiente. Ela será uma derivada analítica da função custo. Use essa função para avaliar o gradiente no ponto $x = 4.5$ e combine o resultado com a taxa de aprendizado para chegar ao próximo passo do processo de gradiente descendente.

```

def gradient(X):
    return (2*X) - 2
x_start = 4.5
learning_rate = 0.75
x_next = x_start - gradient(x_start)*learning_rate
x_next

```

Nota: Não há problema se você não estudou integral e não entender essa parte; só é preciso saber que essa é a função do gradiente. Em algumas aplicações, não é possível calcular uma derivada analítica, logo, é necessário usar aproximação numérica.

Após executar a célula que contém `x_next`, você obterá a saída a seguir:

```
-0.75
```

Esse é o próximo passo do gradiente descendente após $x = 4.5$.

3. Plote o caminho do gradiente descendente, do ponto inicial ao próximo ponto, usando este código:

```
plt.plot(X_poly, y_poly)
plt.plot([x_start, x_next], [cost_function(x_start), cost_function(x_next)], '-o')
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.legend(['Error surface', 'Gradient descent path'])
```

Você obterá esta saída:

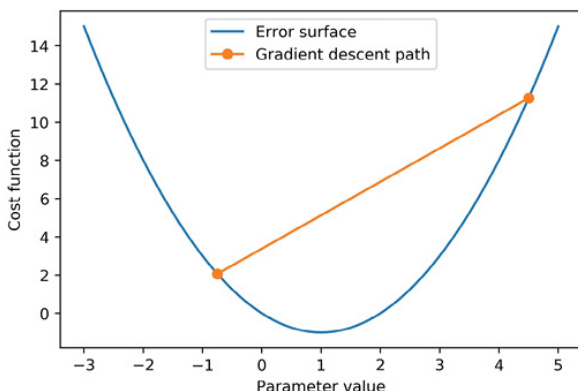


Figura 4.7 – Primeiro passo do caminho do gradiente descendente.

Parece que demos um passo na direção certa. No entanto, está claro que “ultrapassamos” onde queríamos estar. Talvez nossa taxa de aprendizado seja grande demais e consequentemente também estamos dando passos muito grandes. Embora ajustar a taxa de aprendizado seja uma boa ideia para convergirmos em direção a uma solução ótima rapidamente, nesse exemplo podemos apenas continuar ilustrando o resto do processo. Podemos precisar dar mais alguns passos. Na prática, o gradiente descendente continua até o tamanho dos passos ficar muito pequeno ou a alteração na função custo tornar-se bem menor (você pode especificar o quanto deseja que ela fique menor usando o argumento `tol` na regressão logística do `scikit-learn`), indicando que estamos “suficientemente próximos” de uma boa solução – isto é, de um **mínimo local** da função custo. Para esse exemplo, daremos um total de 14 passos, ou **iterações**, além do palpite inicial (observe que você também pode definir o número máximo de iterações no `scikit-learn` com `max_iter`).

4. Execute 14 iterações para convergir em direção ao mínimo local da função custo usando o fragmento de código a seguir (repare que `iterations` é igual a 15, mas o endpoint não foi incluído na chamada

a `range()`):

```
iterations = 15
x_path = np.empty(iterations,)
x_path[0] = x_start
for iteration_count in range(1,iterations):
    derivative = gradient(x_path[iteration_count-1])
    x_path[iteration_count] = x_path[iteration_count-1] - (derivative*learning_rate)
x_path
```

Você obterá a saída a seguir:

```
array([ 4.5, -0.75, 1.875, 0.5625, 1.21875,
        0.890625, 1.0546875, 0.97265625, 1.01367188, 0.99316406,
        1.00341797, 0.99829102, 1.00085449, 0.99957275, 1.00021362])
```

Pelos valores resultantes do processo de gradiente descendente, parece que (no final) chegamos muito perto (1.00021362) da solução ótima (1).

5. Plote o caminho do gradiente descendente usando este código:

```
plt.plot(X_poly, y_poly)
plt.plot(x_path, cost_function(x_path), '-o')
plt.xlabel('Parameter value')
plt.ylabel('Cost function')
plt.legend(['Error surface', 'Gradient descent path'])
```

Você verá a seguinte saída:

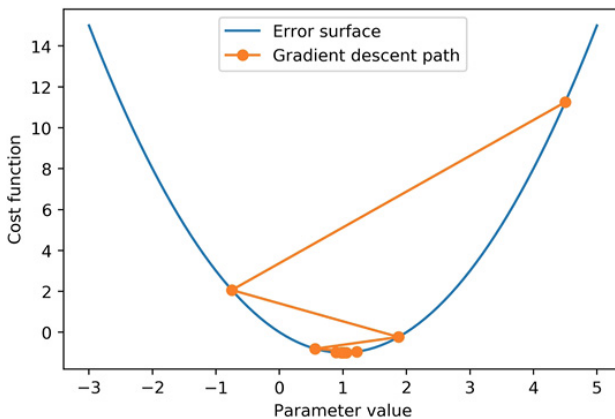


Figura 4.8 – Caminho do gradiente descendente.

Recomendamos que você repita o procedimento anterior com diferentes taxas de aprendizado para ver como elas afetam o caminho do gradiente descendente. Com a taxa de aprendizado certa, é possível convergir para uma solução altamente precisa com muita rapidez. Embora a escolha da taxa de aprendizado seja importante em diferentes aplicações de machine learning, para a

regressão logística geralmente o problema é muito fácil de resolver e não é necessário selecionar uma taxa de aprendizado no scikit-learn.

À medida que você testou diferentes taxas de aprendizado, notou o que aconteceu quando a taxa era maior do que um? Nesse caso, o passo dado em direção à diminuição do erro é grande demais e acabamos obtendo um erro maior. Esse problema pode aumentar e levar o processo de gradiente para longe da região de erro mínimo. Por outro lado, se o tamanho do passo for pequeno demais, podemos demorar para encontrar a solução desejada.

Suposições da regressão logística

Já que é um modelo estatístico clássico, semelhante ao teste F e à correlação de Pearson que examinamos, a regressão logística faz certas suposições sobre os dados. Embora não seja necessário seguir cada uma dessas suposições no sentido mais estrito possível, é bom conhecê-las. Dessa forma, se um modelo de regressão logística não estiver tendo um bom desempenho, você pode investigar e descobrir a razão usando seu conhecimento da situação ideal na qual a regressão funcionaria bem. Você pode encontrar outras listas de suposições específicas obtidas em fontes distintas, mas as descritas aqui são amplamente aceitas.

As características são lineares no logaritmo das chances

Conhecemos essa suposição no capítulo anterior, ou seja, no *Capítulo 3, Detalhes da regressão logística e exploração de características*. A regressão logística é um modelo linear, logo, só funciona quando as características conseguem descrever uma tendência linear no logaritmo das chances. Especificamente, a regressão logística não captura interações, características polinomiais ou a discretização de características por conta própria. No entanto, você pode especificar tudo isso como “novas características” – ainda que tenham sido obtidas por engenharia de características existentes.

Lembre-se de que vimos no capítulo anterior que a característica mais importante da exploração de características univariada, PAY_1, não foi considerada linear no logaritmo das chances.

Não há multicolinearidade de características

Multicolinearidade significa que as características estão

correlacionadas. A pior violação dessa suposição ocorre quando as características estão perfeitamente correlacionadas, como quando uma característica é idêntica a outra ou é igual a outra multiplicada por uma constante. Podemos investigar a correlação de características usando a plotagem de correlação com a qual nos familiarizamos na seleção de características univariada. Carregaremos os dados e recriaremos a plotagem, da mesma forma que fizemos anteriormente:

Nota: Ajuste o caminho do código a seguir para o local em que você salvou os dados limpos do *Capítulo 1, Exploração e limpeza de dados*.

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
features_response = df.columns.tolist()
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school', 'none',
                   'others', 'university']
features_response = [item for item in features_response if item not in items_to_remove]
corr = df[features_response].corr()
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
sns.heatmap(corr,
            xticklabels=corr.columns.values,
            yticklabels=corr.columns.values,
            center=0)
```

A plotagem pode visualizada na Figura 4.9.

Podemos ver pela plotagem como se dá uma correlação perfeita (isto é, $p=1$): já que cada característica e a variável de resposta têm correlação igual a 1 com elas próprias, podemos ver que uma correlação igual a 1 tem uma cor creme clara. Pela barra de cores, que não inclui -1, sabemos que não há correlações com esse valor.

Os exemplos mais claros de preditores correlacionados nos dados de nosso estudo de caso são as características BILL_AMT. Faz sentido que as faturas sejam semelhantes todo mês, por exemplo, nesses dois tipos de contas: uma conta que normalmente tenha saldo zero ou uma que tenha saldo alto e demore um pouco para ser paga. Há características BILL_AMT perfeitamente correlacionadas? A resposta é não, o que podemos confirmar na *Figura 4.9*. Logo, embora talvez essas características não forneçam muitas informações independentes, não vemos razão para removê-las nesse momento.

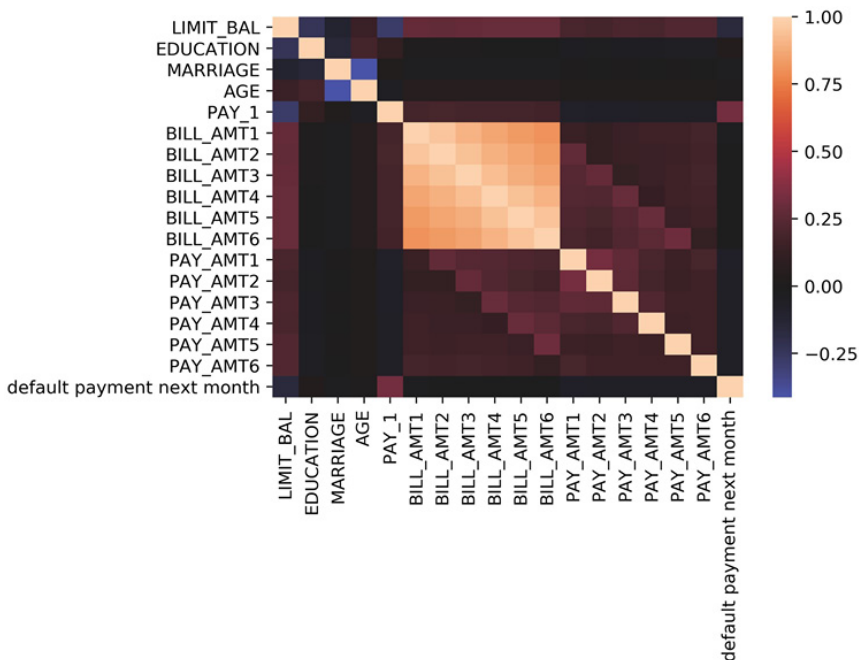


Figura 4.9 – Plotagem de correlação de características e da resposta.

Independência das observações

Essa é uma suposição comum nos modelos estatísticos clássicos, inclusive na regressão linear. Nesse caso, supõe-se que as observações (ou amostras) sejam independentes. Isso faz sentido com os dados do estudo de caso? Queríamos confirmar com nosso cliente se a mesma pessoa pode ter várias contas de crédito para o dataset em questão e considerar o que fazer dependendo do quanto isso for comum. Suponhamos que disséssemos que em nossos dados cada conta de crédito pertence a uma única pessoa, logo, podemos supor independência de observações nesse aspecto.

Entre as diferentes áreas de dados, algumas violações comuns na independência das observações são as seguintes:

- **Autocorrelação espacial** das observações; por exemplo, em fenômenos naturais como os que envolvem o tipo de solo, em que as observações que estão geograficamente mais próximas podem ser semelhantes umas às outras.
- **Autocorrelação temporal** das observações, que pode ocorrer em dados de séries temporais. Geralmente se supõe que as observações do momento atual estejam correlacionadas com as de momentos mais recentes.

No entanto, essas questões não são relevantes para os dados de nosso estudo de caso.

Não há valores atípicos

Os outliers (valores atípicos) são observações em que o valor da(s) característica(s) ou da resposta está muito longe da maioria dos dados ou é diferente de alguma outra forma. Um termo mais apropriado para a observação de valor atípico em uma característica é ponto alto de alavancagem, já que geralmente o termo “outlier” é aplicado à variável de resposta. Porém, em nosso problema de classificação binária, não é possível haver um valor atípico para a variável de resposta, porque ela só pode assumir valores entre 0 e 1. Na prática, os dois termos costumam ser usados para as características.

Para ver por que esses tipos de pontos podem ter um efeito adverso sobre os modelos lineares como um todo, examine esses dados lineares sintéticos e a linha de melhor ajuste que resulta da regressão linear:

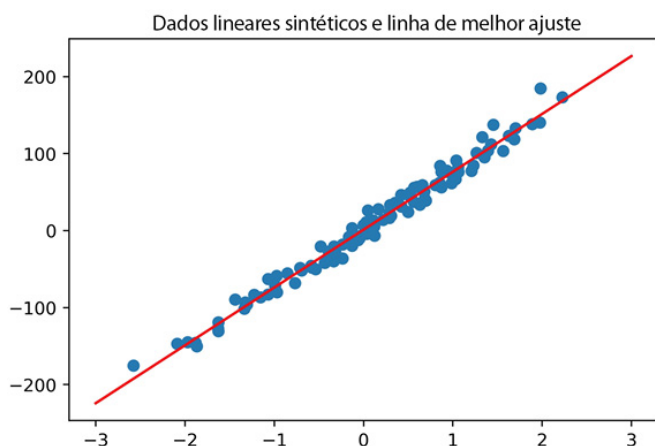


Figura 4.10 – Dados lineares “comportados” e um ajuste de regressão.

Aqui, o modelo parece ser intuitivamente um bom ajuste para os dados. No entanto, e se um valor atípico fosse adicionado para uma característica? Para ilustrar esse caso, adicionaremos um ponto com valor x que é muito diferente do da maioria das observações e valor y que está em um intervalo semelhante ao do resto. Podemos ver então a linha de regressão resultante:

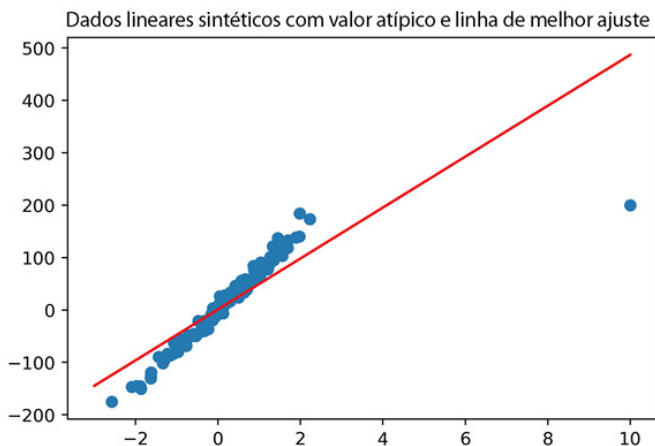


Figura 4.11 – Plotagem mostrando o que acontece quando um valor atípico é incluído.

Devido à presença de um único ponto alto de alavancagem, o ajuste do modelo de regressão para todos os dados não é mais uma boa representação da maioria deles. Isso mostra o efeito potencial de um único ponto de dados sobre os modelos lineares, principalmente se esse ponto não seguir a mesma tendência do resto dos dados.

Há métodos para a manipulação de valores atípicos. Contudo, uma pergunta mais básica que devemos fazer é “Dados como esses são realistas?”. Se os dados não parecerem corretos, é uma boa ideia perguntar ao cliente se os valores atípicos são críveis. Se não forem, devemos excluí-los. No entanto, se representarem dados válidos, modelos não lineares ou outros métodos devem ser usados.

Com os dados de nosso estudo de caso, não observamos valores atípicos nos histogramas que plotamos durante a exploração de características. Logo, não precisamos nos preocupar.

Quantas características você deve incluir?

Em vez de uma suposição, essa questão está mais para uma orientação sobre a construção de modelos. Não há uma regra clara que defina quantas características devemos incluir em um modelo de regressão logística. Porém, uma regra prática comum é a “regra de 10”, que define que para cada 10 ocorrências da classe de resultado mais rara, 1 característica pode ser adicionada ao modelo. Logo, por exemplo, em um problema binário de regressão logística com 100 amostras, se a proporção da classe tiver 20% de resultados positivos e 80% de resultados negativos, haverá um total de apenas 20 resultados positivos e só 2 características devem ser usadas no modelo. Uma

“regra de 20” também já foi sugerida, sendo um limite mais restrito para o número de características a serem incluídas (1 característica em nosso exemplo).

Outro ponto que devemos considerar no caso das características binárias, como as que resultam da codificação one-hot, é quantas amostras terão um valor positivo para essa característica. Se a característica estiver muito desbalanceada, em outras palavras, com poucas amostras contendo 1 ou 0, talvez não faça sentido incluí-la no modelo.

Para os dados do estudo de caso, temos sorte por haver um número relativamente grande de amostras e características relativamente balanceadas, logo, não temos essas preocupações.

Motivação para a regularização: O trade-off entre viés e variância

Podemos estender o modelo básico de regressão logística sobre o qual aprendemos usando um conceito poderoso conhecido como **shrinkage** ou **regularização**. Na verdade, todas as regressões logísticas que ajustamos até agora no scikit-learn empregaram algum nível de regularização. Isso ocorre porque essa é uma opção padrão que é definida no objeto de modelo de regressão logística; no entanto, até o momento, a ignoramos.

À medida que você conhecer esses conceitos com mais detalhes, também se familiarizará com alguns conceitos básicos de machine learning: **overfitting**, **underfitting** e o trade-off entre viés e variância. Diz-se que um modelo está causando o sobreajuste (overfit) dos dados de treinamento quando seu desempenho com esses dados (por exemplo, a ROC AUC) é substancialmente melhor do que o desempenho com um conjunto de teste. Em outras palavras, o bom desempenho com o conjunto de treinamento não é generalizado para o conjunto de teste desconhecido. Começamos a discutir esses conceitos no *Capítulo 2, Introdução ao scikit-learn e avaliação do modelo*, quando fizemos a distinção entre os resultados (scores) do modelo no treinamento e em teste.

Quando um modelo está causando o sobreajuste dos dados de treinamento, diz-se que ele tem alta **variância**. Isso significa que, qualquer que seja a variabilidade existente nos dados de treinamento, o modelo a assimilou bem – na verdade, bem demais. Logo, teremos um ótimo resultado no treinamento do modelo. No entanto, quando

esse modelo for usado para fazer previsões de dados novos e desconhecidos, o desempenho será mais baixo na fase de teste. O overfitting tem mais probabilidade de ocorrer nas circunstâncias a seguir:

1. Há um número maior de características disponíveis em relação ao número de amostras. Especificamente, pode haver tantas características que fica difícil inspecionar todas diretamente, como fizemos com os dados do estudo de caso.
2. Um modelo complexo, isto é, mais complexo que a regressão logística, é usado. Isso inclui modelos como as florestas aleatórias, combinações de modelos, ou redes neurais.

Nessas circunstâncias, o modelo tem a oportunidade de desenvolver **hipóteses** mais complexas sobre o relacionamento entre as características e a variável de resposta nos dados de treinamento durante seu ajuste, tornando mais provável a ocorrência de overfitting.

Por outro lado, se um modelo não estiver ajustando os dados de treinamento muito bem, isso é conhecido como underfitting, e diz-se que o modelo tem **viés** alto.

Podemos examinar as diferenças entre underfitting, overfitting e a situação ideal, que é intermediária, ajustando modelos polinomiais com alguns dados hipotéticos:

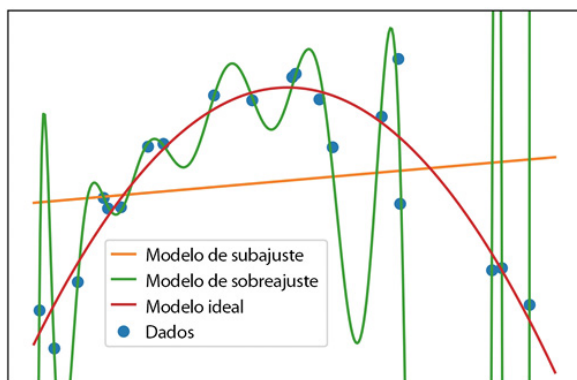


Figura 4.12 – Dados quadráticos com modelos de subajuste, sobreajuste e ideal.

Na *Figura 4.12*, podemos ver que a inclusão de menos características, nesse caso um modelo linear de y com apenas duas características, uma inclinação e a interceptação, claramente não é uma boa representação dos dados. Isso é conhecido como modelo de subajuste.

No entanto, se incluirmos características demais, isto é, com muitos termos polinomiais de alto grau, como x^2 , x^3 , x^4 ,... x^{10} , poderemos ajustar os dados de treinamento quase que perfeitamente. Porém, isso não é necessariamente algo bom. Quando examinamos os resultados desse modelo entre os pontos de dados de treinamento, nos quais novas previsões podem ter de ser feitas, vemos que ele é instável e talvez não forneça previsões confiáveis para dados que não estejam no conjunto de treinamento. Sabemos disso com base em um conhecimento intuitivo do relacionamento entre as características e a variável de resposta, que podemos obter pela inspeção dos dados brutos.

Já que sabemos que esses dados foram gerados por um polinômio de segundo grau (isto é, quadrático), podemos encontrar facilmente o modelo ideal ajustando um polinômio de segundo grau para os dados de treinamento, como mostrado na *Figura 4.12*. Em geral, entretanto, não sabemos qual é a formulação do modelo ideal com antecedência. Logo, temos de comparar os resultados de treinamento e de teste para avaliar se um modelo está causando overfitting ou underfitting.

Em alguns casos, pode ser desejável introduzir algum viés no processo de treinamento do modelo, principalmente se isso diminuir o overfitting e melhorar o desempenho para dados novos e desconhecidos. Dessa forma, talvez possamos nos beneficiar do trade-off entre viés e variância para melhorar o modelo. Podemos usar métodos de **regularização** para fazê-lo. Além disso, talvez possamos usar esses métodos na **seleção de variáveis** como parte do processo de modelagem. Usar um modelo preditivo para selecionar variáveis é uma alternativa à seleção de características univariada que examinamos. Começaremos a fazer testes com esses conceitos no exercício a seguir.

Exercício 17: Gerando e modelando dados de classificação sintéticos

Neste exercício, observaremos o overfitting na prática usando um dataset sintético. Suponhamos que você tivesse recebido um dataset de classificação binária com muitas características candidatas (200) e não tivesse tempo para examinar todas individualmente. É possível que algumas dessas características estejam altamente correlacionadas ou tenham algum outro tipo de relação. No entanto, com a existência de tantas variáveis, pode ser difícil explorar todas. Além disso, o dataset tem relativamente poucas amostras: apenas 1.000. Geraremos

esse dataset desafiador usando um recurso do scikit-learn que permite criar datasets sintéticos para explorações conceituais como essa. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook. Eles podem ser encontrados em <http://bit.ly/2ZAy2Pr>.

1. Importe as classes `make_classification`, `train_test_split`, `LogisticRegression` e `roc_auc_score` usando este código:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score
```

Observe que primeiro importamos várias classes familiares do scikit-learn, além de uma nova que ainda não vimos: `make_classification`. Essa classe faz exatamente o que seu nome sugere – cria dados para um problema de classificação. Usando os diversos argumentos de palavra-chave, você pode especificar quantas amostras e características serão incluídas e quantas classes a variável de resposta terá. Também há um conjunto de outras opções que controlam com que “facilidade” resolveremos o problema.

Nota: Para obter mais informações, consulte https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html. É bom ressaltar que selecionamos opções que resultam em um problema razoavelmente fácil de resolver, com a introdução de alguns obstáculos. Em outras palavras, esperamos um modelo de alto desempenho, mas teremos de fazer algum esforço para obtê-lo.

2. Gere um dataset com duas variáveis, `x_synthetic` e `y_synthetic`, 200 características candidatas e 1.000 amostras usando o código a seguir:

```
X_synthetic, y_synthetic = \ make_classification(n_samples=1000, n_features=200,
    n_informative=3, n_redundant=10, n_repeated=0, n_classes=2,
    n_clusters_per_class=2, weights=None, flip_y=0.01, class_sep=0.8,
    hypercube=True, shift=0.0, scale=1.0, shuffle=True, random_state=24)
```

3. Examine a dimensão do dataset usando o seguinte código:

```
print(X_synthetic.shape, y_synthetic.shape)
```

Você obterá esta saída:

```
(1000, 200) (1000,)
```


Observe que geramos um dataset quase perfeitamente balanceado: com um equilíbrio de classes próximo de 50/50. Também é importante notar que geramos todas as características de modo que elas tenham a mesma escala – isto é, uma média igual a 0 com desvio padrão 1. Certificar-se de que as características estejam na mesma escala, ou tenham aproximadamente o mesmo intervalo de valores, é um ponto-chave para o uso de métodos de regularização – e veremos por que posteriormente. Se as características de um dataset bruto estiverem em escalas muito diferentes, é aconselhável normalizá-las para que fiquem na mesma escala. O scikit-learn tem uma funcionalidade que facilita isso; aprenderemos sobre ela na atividade do fim deste capítulo.

4. Plote as primeiras características como histogramas para mostrar que o intervalo de valores é o mesmo usando o código a seguir:

```
for plot_index in range(4):  
    plt.subplot(2,2,plot_index+1)  
    plt.hist(X_synthetic[:,plot_index])  
    plt.title('Histogram for feature {}'.format(plot_index+1))  
plt.tight_layout()
```

Você obterá a seguinte saída:

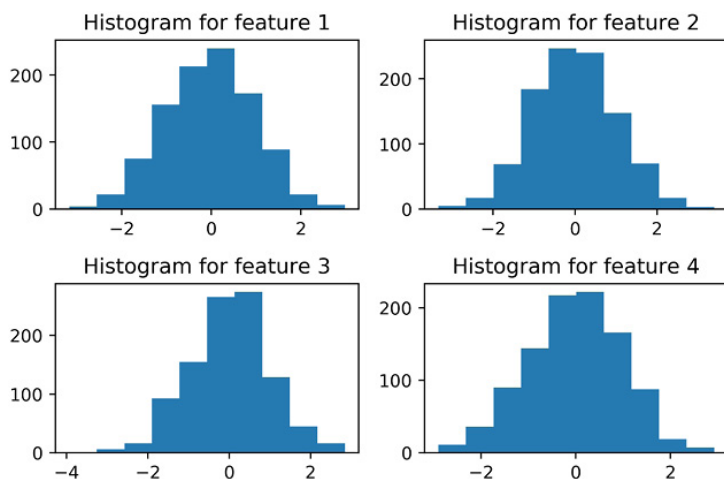


Figura 4.13 – Histogramas das 4 primeiras características das 200 existentes.

Já que geramos esse dataset, não precisamos examinar diretamente todas as 200 características para verificar se elas estão na mesma escala. Logo, que preocupações teríamos com o dataset? Os dados estão balanceados em termos das frações de classes da variável de resposta, portanto, não precisamos diminuir ou aumentar as

amostras, ou usar outros métodos úteis para dados desbalanceados. E quanto aos relacionamentos entre as características propriamente ditas e entre elas e a variável de resposta? Há vários desses relacionamentos e é um desafio investigar todos diretamente. De acordo com nossa “regra prática”, 200 características é um número excessivo (principalmente se usarmos a “regra de 10”). Temos 500 observações na classe mais rara, logo, por essa regra não devíamos ter mais do que 50 características. É possível que com tantas características, das quais não temos uma boa ideia da qualidade, o procedimento de treinamento do modelo cause overfitting.

5. Divida os dados em conjuntos de treinamento e de teste usando uma razão de 80/20 e instancie um objeto de modelo de regressão logística com este código:

```
X_syn_train, X_syn_test, y_syn_train, y_syn_test = train_test_split(
X_synthetic, y_synthetic,
test_size=0.2, random_state=24)
lr_syn = LogisticRegression(solver='liblinear', penalty='l1', C=1000, random_state=1)
lr_syn.fit(X_syn_train, y_syn_train)
```

Observe que estamos especificando algumas opções novas no modelo de regressão logística às quais até agora não demos atenção. Em primeiro lugar, especificamos o argumento `penalty` como `l1`. Isso significa que usaremos a **regularização L1**, que também é conhecida como **regularização lasso**. Discutiremos a definição matemática em breve. Em segundo lugar, repare que definimos o parâmetro `C` como igual a 1.000. `C` é o “inverso da força de regularização”, de acordo com a documentação do `scikit-learn`

(https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Ou seja, valores maiores de `C` correspondem a menos regularização. Ao selecionar um número relativamente alto, estamos usando pouca regularização. O valor padrão de `C` é 1. Logo, na verdade não estamos usando muita regularização aqui; em vez disso, estamos simplesmente nos familiarizando com as opções de uso. Para concluir, estamos usando o solver `liblinear`, que já empregamos no passado. Embora estejamos utilizando dados na mesma escala, é bom ressaltar que, entre as diversas opções que temos disponíveis para solvers, `liblinear` é “robusto para dados de escalas diferentes”. Além disso, `liblinear` é uma das duas únicas opções de solver que dão suporte à penalização L1 – a outra é `saga`.

Nota: Há mais informações sobre os solvers disponíveis em

6. Ajuste o modelo de regressão logística com os dados de treinamento usando o código a seguir:

Nota: Esse processo traz de volta todas as opções que indicamos ao instanciar o modelo e os padrões que não definimos. Incluímos essas saídas nas instâncias relevantes do fragmento de código.

```
lr_syn.fit(X_syn_train, y_syn_train)
```

Aqui está a saída:

```
LogisticRegression(C=1000, class_weight=None, dual=False, fit_intercept=True,
    intercept_scaling=1, max_iter=100, multi_class='warn',
    n_jobs=None, penalty='l1', random_state=1, solver='liblinear',
    tol=0.0001, verbose=0, warm_start=False)
```

7. Calcule o resultado do treinamento usando este código, primeiro obtendo as probabilidades previstas e depois encontrando a ROC AUC:

```
y_syn_train_predict_proba = lr_syn.predict_proba(X_syn_train)
roc_auc_score(y_syn_train, y_syn_train_predict_proba[:,1])
```

A saída deve ser:

```
0.9420000000000001
```

8. Calcule o resultado do teste da mesma maneira que o resultado do treinamento foi obtido:

```
y_syn_test_predict_proba = lr_syn.predict_proba(X_syn_test)
roc_auc_score(y_syn_test, y_syn_test_predict_proba[:,1])
```

A saída é:

```
0.8074807480748075
```

A partir desses resultados, fica claro que o modelo de regressão logística sobreajustou os dados. Isto é, o resultado da ROC AUC com os dados de treinamento é substancialmente maior que o dos dados de teste.

Regularização lasso (L1) e ridge (L2)

Antes de aplicar uma regularização a um modelo de regressão logística, vamos tentar entender o que é regularização e como ela funciona. As duas técnicas de regularização de modelos de regressão logística do scikit-learn são chamadas de **lasso** (também conhecida como regularização **L1**) e **ridge** (a regularização **L2**). Ao instanciar o objeto de modelo a partir de uma classe do scikit-learn, você pode

selecionar $\text{penalty} = 'l1'$ ou $'l2'$. Essas são as chamadas “penalizações”, porque o efeito da regularização é a inclusão de uma penalização, ou de um custo, pela existência de valores maiores para os coeficientes em um modelo de regressão logística ajustado.

Como já vimos, os coeficientes de um modelo de regressão logística descrevem o relacionamento entre o logaritmo das chances da resposta e cada uma das características. Logo, se o valor de um coeficiente for particularmente alto, uma pequena alteração nessa característica produzirá um efeito maior sobre a previsão. Quando um modelo está sendo ajustado, e está aprendendo o relacionamento entre as características e a variável de resposta, ele pode começar a detectar “ruído” nos dados. Vimos isso anteriormente na *Figura 4.12*: se houver muitas características disponíveis no ajuste de um modelo, e não houver “limites” para os valores que seus coeficientes podem assumir, o processo de ajuste pode tentar descobrir relacionamentos entre as características e a variável de resposta que não serão generalizados para dados novos. Dessa forma, o modelo fica em sintonia com o ruído aleatório imprevisível que acompanha os dados imperfeitos do mundo real. Infelizmente, isso só serve para aumentar a facilidade de o modelo prever dados de treinamento, o que não é nosso objetivo final. Portanto, precisamos tentar erradicar esses relacionamentos espúrios do modelo.

A regularização lasso e a regularização ridge usam diferentes formulações matemáticas para atingir esse objetivo. Esses métodos funcionam fazendo alterações na função custo usada no ajuste do modelo, que introduzimos anteriormente como função de perda logarítmica. A regularização lasso usa o que é chamado de **norma-1** (daí o termo L1):

$$\text{perda logarítmica com penalização lasso} = \sum_{j=1}^m |\sigma_j| + \frac{C}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figura 4.14 – Equação de perda logarítmica com penalização lasso.

A norma-1 $\sum_{j=1}^m |\sigma_j|$ é apenas a soma dos valores absolutos de todos os coeficientes das m diferentes características. O valor absoluto é usado porque um coeficiente alto na direção positiva ou negativa pode contribuir para o overfitting. O que mais é diferente entre essa função custo e a função de perda logarítmica que vimos anteriormente? Bem, agora há um fator C que é multiplicado pelo numerador da fração na frente da soma da função de perda logarítmica ao longo de todas as n amostras de treinamento. É o “inverso da força de regularização”

como descrito na documentação do scikit-learn (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html). Já que esse fator está na frente do termo da função custo que calcula o erro na previsão, em vez do termo que executa a regularização, torná-lo maior faz com que o erro na previsão seja mais importante na função custo, enquanto a regularização passa a ser menos importante. Resumindo, *valores maiores para C levam a menos regularização* na implementação do scikit-learn.

A regularização L2, ou ridge, é semelhante à L1, exceto por, em vez de usar a soma de valores absolutos dos coeficientes, empregar a soma de seus quadrados, a chamada **norma-2**:

$$\text{perda logarítmica com penalização ridge} = \sum_{j=1}^m \sigma_j^2 + \frac{C}{n} \sum_{i=1}^n -(y_i \log(p_i) + (1 - y_i) \log(1 - p_i))$$

Figura 4.15 – Equação de perda logarítmica com penalização ridge.

Observe que, se você examinar as funções custo da regressão logística na documentação do scikit-learn, a forma específica é diferente da usada aqui, mas a ideia geral é semelhante. Além disso, quando você se sentir confortável com os conceitos das penalizações lasso e ridge, é bom ressaltar que há um método de regularização adicional chamado **rede elástica**, que é uma combinação tanto da regularização lasso quanto da regularização ridge.

Por que há duas formulações para a regularização?

Uma das duas pode fornecer resultados melhores para você, logo, é preciso testá-las. Há outra diferença importante em seus resultados: a penalização L1 também executa a seleção de características, além da regularização. Ela faz isso configurando alguns valores de coeficientes com zero durante o processo de regularização, removendo características do modelo. A regularização L2 torna os valores dos coeficientes menores, mas não os elimina totalmente. Não são todas as opções de solver do scikit-learn que dão suporte às duas regularizações, logo, você terá de selecionar um solver apropriado para a técnica de regularização que quiser usar.

Nota: Embora os detalhes matemáticos de por que isso ocorre não façam parte do escopo deste livro, para ver uma explicação completa do tópico e fazer uma consulta mais geral, recomendamos o recurso de fácil leitura (e gratuito) *An Introduction to Statistical Learning*, de Gareth James e colaboradores. Especificamente, leia a página 222 da sétima

impressão corrigida, que estava disponível quando este texto foi escrito, para ver um gráfico útil sobre a diferença entre as regularizações L1 e L2.

Interceptações e regularização

Não entramos em uma discussão mais detalhada sobre as interceptações e apenas as estimamos em nossos modelos lineares junto com os coeficientes que acompanham cada característica. Você precisa usar uma interceptação? A resposta é provavelmente sim, até desenvolver um conhecimento avançado dos modelos lineares e ter certeza de que em um caso específico não deve usá-la. No entanto, esses casos existem, por exemplo, em uma regressão linear em que as características e a variável de resposta tenham sido normalizadas para obtenção de uma média igual a zero.

As interceptações não acompanham nenhuma característica específica. Logo, não faz muito sentido regularizá-las, já que não devem contribuir para o overfitting. Observe que, no termo de penalização da regularização L1, a soma começa com $j = 1$, e o mesmo ocorre para a regularização L2: pulamos σ_0 , que é o termo da interceptação.

Essa é a situação ideal: não regularizar a interceptação. No entanto, devido à maneira como os diferentes solvers são implementados no scikit-learn, o solver liblinear faz isso. Porém, há uma opção `intercept_scaling` que você pode fornecer para a classe de modelo a fim de neutralizar esse efeito. Não ilustramos essa técnica aqui porque, embora teoricamente seja incorreta, na prática regularizar a interceptação não costuma afetar muito a qualidade preditiva do modelo.

Escalonamento e regularização

Como observado no exercício anterior, é uma prática recomendável **escalonar** os dados para que todas as características tenham aproximadamente o mesmo intervalo de valores antes do uso da regularização. Isso ocorre porque todos os coeficientes estarão sujeitos à mesma penalização na função custo. Se o intervalo de valores de uma característica específica, como `LIMIT_BAL` em nosso dataset, for muito maior do que o de outras características, como `PAY_1`, pode ser melhor termos um valor maior para o coeficiente de `PAY_1` e um menor para o de `LIMIT_BAL` a fim de nivelarmos seus efeitos na mesma escala da combinação linear de características e coeficientes usados na previsão do modelo. Normalizar todas as características

antes de usar a regularização evita complicações como essa que surgem simplesmente de diferenças na escala.

Na verdade, pode ser necessário escalonar os dados, dependendo do solver que você usar. As diferentes variações no processo de gradiente descendente disponível no scikit-learn podem ou não conseguir trabalhar com dados não escalonados.

Importância de selecionar o solver certo

Como vimos, os diferentes solvers disponíveis para a regressão logística no scikit-learn têm comportamentos distintos com relação ao seguinte:

- Se darão suporte tanto à regularização L1 quanto à regularização L2
- Como tratarão a interceptação durante a regularização
- Como lidarão com dados não escalonados

Nota: Há mais diferenças. Uma tabela útil que compara essas e outras peculiaridades está disponível em https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression. Você pode usá-la para definir que solver é apropriado para seu problema.

Nesta seção aprendemos os fundamentos matemáticos das regularizações lasso e ridge. *Esses métodos operam reduzindo os valores dos coeficientes em direção a 0, e, no caso do lasso, definindo alguns coeficientes com exatamente zero e executando a seleção de características.* Observe que em nosso exemplo de overfitting da *Figura 4.12*, se o modelo complexo e de sobreajuste tivesse alguns coeficientes reduzidos em direção a 0, ele chegaria mais perto do modelo ideal, que tem menos coeficientes.

Aqui está uma plotagem de um modelo de regressão regularizado, usando as mesmas características polinomiais de alto grau do modelo de sobreajuste, mas com penalização ridge:

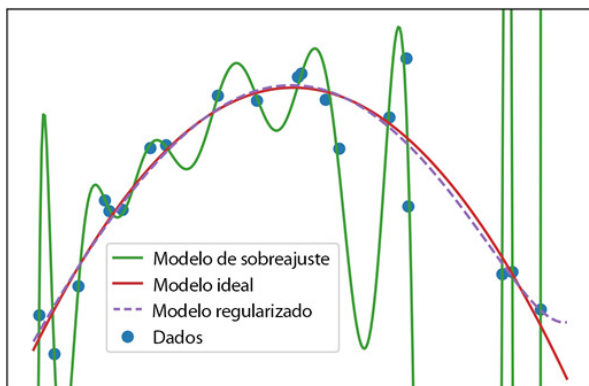


Figura 4.16 – Modelo de sobreajuste e modelo regularizado usando as mesmas características.

O modelo regularizado parece semelhante ao modelo ideal. Observe, porém, que não é recomendado extrapolá-lo. Podemos ver que o modelo regularizado começa a aumentar em direção ao lado direito da *Figura 4.16*. Esse aumento deve ser considerado suspeito, já que não há nada nos dados de treinamento que deixe claro que isso é esperado. Esse exemplo mostra que a *extrapolação das previsões do modelo para fora do intervalo dos dados de treinamento não é recomendada*. No entanto, podemos deduzir da *Figura 4.16* que, mesmo se não conhecermos o modelo que foi usado para gerar esses dados sintéticos (já que normalmente não conhecemos o processo de geração de dados em trabalhos de modelagem preditiva do mundo real), podemos usar a regularização para reduzir o efeito do overfitting quando um grande número de características candidatas estiver disponível.

Modelos e seleção de características

A regularização L1 é uma maneira de usarmos um modelo, como o de regressão logística, para executar a seleção de características. Outros métodos seriam os de **seleção stepwise** forward ou backward. A ideia geral existente por trás desses métodos é a seguinte: no caso da **seleção forward**, as características são adicionadas ao modelo uma a uma e o desempenho do teste fora da amostra (out-of-sample) é observado durante o processo. A cada iteração, a inclusão de todas as características possíveis do pool de candidatas é considerada e a que resultar no maior aumento de desempenho fora da amostra é selecionada. Quando a inclusão de características adicionais deixa de produzir o efeito de melhorar o desempenho do modelo, não é preciso adicionar mais candidatas. Na **seleção backward**, começamos com todas as características do modelo e determinamos qual devemos

remover: nesse caso, a que resultar na menor diminuição de desempenho do teste fora da amostra. Você pode continuar removendo características dessa forma até o desempenho começar a diminuir consideravelmente.

Validação cruzada: Selecionando o parâmetro de regularização e outros hiperparâmetros

Talvez você queira usar a regularização para diminuir o overfitting que observamos quando tentamos modelar os dados sintéticos no *Exercício 17, Gerando e modelando dados de classificação sintéticos*. A questão é como selecionar o parâmetro de regularização, C ? C é um exemplo de **hiperparâmetro** de um modelo. Os hiperparâmetros são diferentes dos parâmetros que são estimados quando um modelo é treinado, como os coeficientes e a interceptação de uma regressão logística. Em vez de serem estimados por um procedimento automatizado como os parâmetros, os hiperparâmetros são inseridos diretamente pelo usuário como argumentos de palavra-chave, normalmente na instanciação da classe de modelo. No entanto, como saber que valores escolher?

Os hiperparâmetros são mais difíceis de estimar que os parâmetros. Isso ocorre porque fica a cargo do cientista de dados determinar qual é o melhor valor em vez de permitirmos que um algoritmo de otimização o encontre. No entanto, é possível selecionar programaticamente valores de hiperparâmetros, o que pode ser visto como um tipo de procedimento de otimização. Na prática, no caso do parâmetro de regularização C , normalmente isso é feito pelo ajuste do modelo em um conjunto de dados com um valor específico para C , a determinação do desempenho do treinamento do modelo e a avaliação do desempenho fora da amostra com outro conjunto de dados.

Já estamos familiarizados com o conceito de uso dos conjuntos de treinamento e de teste do modelo. Porém, há uma diferença importante aqui; por exemplo, o que aconteceria se usássemos o conjunto de teste várias vezes para ver o efeito de diferentes valores de C ?

Pode ter lhe ocorrido que, após a primeira vez que você usar o conjunto de teste desconhecido para avaliar o desempenho fora da amostra para um valor específico de C , ele não seja mais um “conjunto de teste desconhecido”. Embora somente os dados de treinamento

tenham sido usados para estimar os parâmetros do modelo (isto é, os coeficientes e a interceptação), agora os dados de teste estão sendo usados para estimar o hiperparâmetro C . Na verdade, os dados de teste tornaram-se dados de treinamento adicionais, no sentido de que estão sendo usados para encontrar um bom valor para o hiperparâmetro.

Logo, é comum dividir os dados em três partes: um conjunto de treinamento, um conjunto de teste e um **conjunto de validação**. O conjunto de validação serve a mais de uma finalidade:

Estimativa de hiperparâmetros

O conjunto de validação pode ser usado repetidamente na avaliação do desempenho fora da amostra com diferentes valores para a seleção de hiperparâmetros.

Comparação de diferentes modelos

Além de encontrar os valores dos hiperparâmetros de um modelo, um conjunto de validação pode ser usado para determinar o desempenho fora da amostra para diferentes modelos; por exemplo, se quiséssemos comparar a regressão logística e a floresta aleatória.

Nota: Melhores práticas de gerenciamento de dados – Como cientista de dados, cabe a você descobrir como dividir seus dados para diferentes tarefas de modelagem preditiva. Em uma situação ideal, é recomendável reservar uma parte dos dados para o fim do processo, após já terem sido selecionados os hiperparâmetros e também o melhor modelo. Esse **conjunto de teste desconhecido** é reservado para a última etapa, quando pode ser usado na avaliação dos últimos esforços de construção do modelo para sabermos como o modelo final se comporta com dados novos desconhecidos. Na reserva do conjunto de teste, é boa prática certificar-se de que as características e a resposta tenham traços semelhantes aos do resto dos dados. Em outras palavras, a fração das classes deve ser a mesma, e a distribuição de características deve ser semelhante. Dessa forma, os dados de teste serão representativos dos dados em que você baseou o modelo.

Embora a validação do modelo seja uma boa prática, ela levanta a questão de se a divisão específica que escolhemos para os dados de treinamento, validação e teste terá algum efeito sobre os resultados que perseguimos. Por exemplo, talvez o relacionamento entre as

características e a variável de resposta seja um pouco diferente no conjunto de teste desconhecido que reservamos, ou no conjunto de validação em comparação com o conjunto de treinamento. É impossível eliminar essa variabilidade, mas podemos usar o método de **validação cruzada** para evitar confiarmos demais em uma divisão específica dos dados.

O scikit-learn fornece funções convenientes para facilitar as análises de validação cruzada. Essas funções desempenham um papel semelhante ao de `train_test_split`, que já usamos, embora o comportamento padrão seja um pouco diferente. Vamos conhecê-las agora; primeiro, importe estas duas classes:

```
from sklearn.model_selection import StratifiedKfold
from sklearn.model_selection import KFold
```

Como em `train_test_split`, temos de especificar que proporção do dataset queremos usar no treinamento versus teste. No entanto, com a validação cruzada (especificamente a **validação cruzada k folds** que foi implementada nas classes que acabamos de importar), em vez de definir uma proporção diretamente, indicaremos apenas quantos folds queremos – isto é, os “**k folds**”. A ideia aqui é que os dados sejam divididos em **k** proporções iguais. Por exemplo, se especificarmos quatro folds, cada fold terá 25% dos dados. Esses folds serão os dados de teste de quatro instâncias separadas de treinamento do modelo, enquanto os restantes 75% de cada fold serão usados para treiná-lo. Nesse procedimento, pontos de dados individuais são usados como dados de treinamento durante um total de $k - 1$ vezes e como dados de teste uma única vez.

Na instanciação da classe, indicaremos o número de folds, se os dados serão ou não embaralhados antes da divisão, e um seed aleatório se quisermos resultados repetíveis entre diferentes execuções:

```
n_folds = 4
k_folds = KFold(n_splits=n_folds, shuffle=False, random_state=1)
```

Aqui, instanciamos um objeto com quatro folds e sem embaralhamento. Usaremos o objeto retornado, que chamamos de `k_folds`, passando para o seu método `.split` as características e os dados da resposta utilizados na validação cruzada. Isso resultará em um **iterador**, o que significa que poderemos percorrer a saída para obter diferentes divisões de dados de treinamento e teste. Se pegarmos os dados de treinamento de nosso problema de classificação sintética, `X_syn_train` e `y_syn_train`, poderíamos percorrer as divisões desta forma:

```
for train_index, test_index in k_folds_iterator.split(X_syn_train, y_syn_train):
```

O iterador retornou os índices das linhas de `X_syn_train` e `y_syn_train`, que podemos usar para indexar os dados. Dentro desse loop `for`, podemos escrever um código para usar os índices na seleção de dados para o treinamento e teste repetidos de um objeto de modelo com diferentes subconjuntos dos dados. Dessa forma, obteremos uma indicação robusta do desempenho fora da amostra com o uso de um valor de hiperparâmetro específico e depois podemos repetir o processo inteiro usando outro valor. Consequentemente, o loop de validação cruzada pode ser **aninhado** dentro de um loop externo para diferentes valores de hiperparâmetros. Mostraremos isso no próximo exercício.

Primeiro queremos saber qual é a aparência dessas divisões. Se plotássemos os índices de `train_index` e `test_index` com cores diferentes, obteríamos algo como:

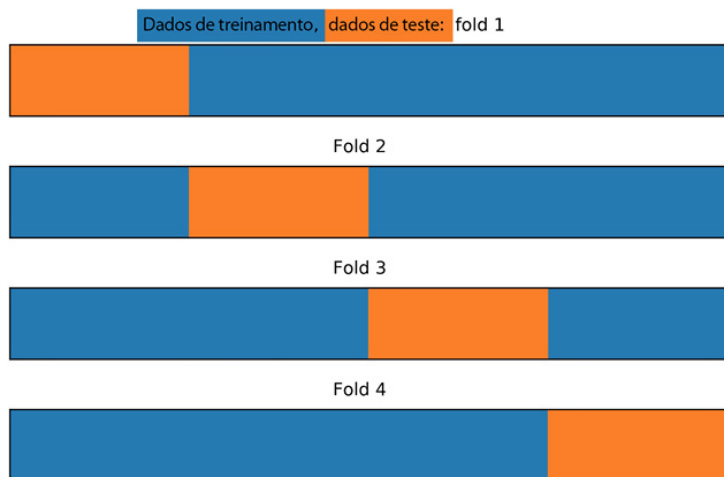


Figura 4.17 – Divisões de treinamento/teste para a validação k-folds com quatro folds e sem embaralhamento.

Aqui, podemos ver que, com as opções que indicamos para a classe `KFold`, o procedimento pegou apenas os primeiros 25% dos dados, de acordo com a ordem das linhas, como o primeiro fold de teste, depois pegou os próximos 25% dos dados para o segundo fold e assim por diante. No entanto, e se quiséssemos folds estratificados? Em outras palavras, e se quiséssemos assegurar que as frações de classes da variável de resposta fossem iguais em cada fold? Embora `train_test_split` permita o uso dessa opção como um argumento de palavra-chave, há uma classe `StratifiedKFold` separada que implementa para a validação cruzada. Nossos dados sintéticos têm um

balanceamento de classes de 50/50 e ele não deve fazer uma grande diferença em nenhum resultado de validação cruzada, mas podemos ilustrar como ficarão as divisões estratificadas desta forma:

```
k_folds = StratifiedKFold(n_splits=n_folds, shuffle=False, random_state=1)
```

Na *Figura 4.18*, podemos ver que houve algum “embaralhamento” (shuffling) entre os diferentes folds. O procedimento moveu amostras entre os folds quando necessário para assegurar que as frações de classes de cada fold sejam iguais.

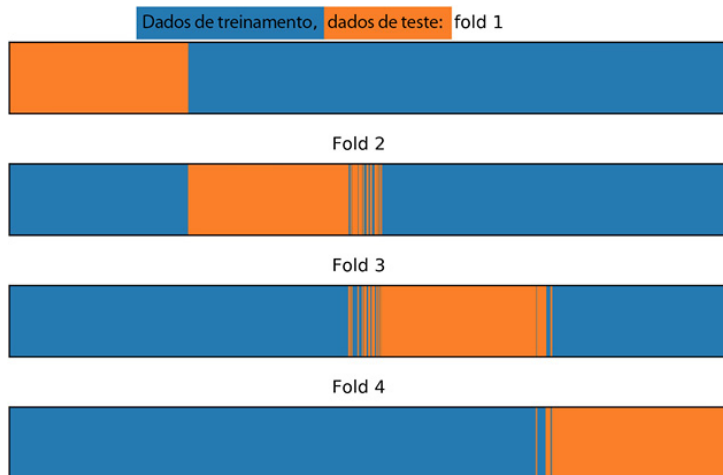


Figura 4.18 – Divisões de treinamento/teste para a validação k-folds estratificada.

Porém, e se quiséssemos embaralhar os dados para selecionar amostras de todo o intervalo de índices para cada fold de teste? Em primeiro lugar, por que faríamos isso? Bem, no que diz respeito aos dados sintéticos que criamos para nosso problema, podemos ter certeza de que eles não estão em nenhuma ordem específica. No entanto, em muitas situações do mundo real, os dados recebidos podem ter sido classificados de alguma forma.

Por exemplo, as linhas dos dados podem ter sido ordenadas pela data em que uma conta foi criada, ou por algum outro tipo de lógica. Logo, pode ser uma boa ideia embaralhar os dados antes de dividir. Assim podemos esperar que a peculiaridade usada na classificação seja consistente em todos os folds. Caso contrário, os dados de diferentes folds podem ter peculiaridades diferentes, possivelmente levando a relacionamentos distintos entre as características e a resposta. Isso pode levar a uma situação em que o desempenho do modelo seja irregular entre os folds. Para “distribuir” os índices de linhas de um

dataset por todos os folds, temos apenas de configurar o parâmetro `shuffle` com `True`:

```
k_folds = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=1)
```

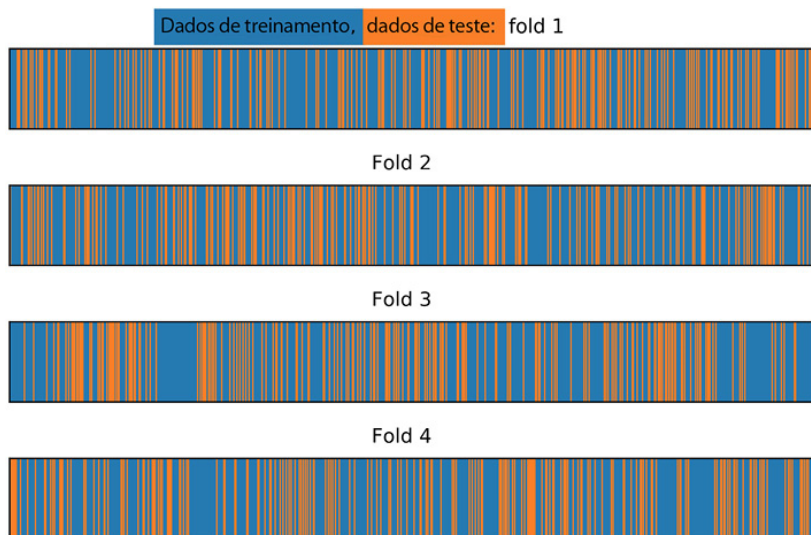


Figura 4.19 – Divisões de treinamento/teste para a validação k-folds estratificada com embaralhamento.

Com o embaralhamento, os folds de teste são distribuídos aleatoriamente, e de maneira bastante uniforme, entre os índices dos dados de entrada.

A validação cruzada k-folds é um método muito usado na ciência de dados. No entanto, a seleção de quantos folds usar vai depender do dataset específico. Usar um número menor de folds significa que a quantidade de dados de treinamento de cada fold será relativamente pequena. Isso aumenta as chances de o modelo sofrer sobreajuste, já que geralmente os modelos funcionam melhor quando treinados com mais dados. É uma boa ideia fazer testes com números de folds diferentes e ver como a média e a variabilidade do resultado da validação k-folds mudam. Os números de folds usados costumam variar de 4 ou 5 a 10.

Se o dataset for muito pequeno, pode ser necessário usar o maior número de dados possível para o treinamento nos folds da validação cruzada. Nesses cenários, você pode usar um método chamado **validação cruzada leave-one-out (LOOCV, leave-one-out cross-validation)**. Na LOOCV, o conjunto de teste de cada fold é composto de uma única amostra. Em outras palavras, haverá um número de folds equivalente ao número de amostras nos dados de treinamento. A

cada iteração, o modelo será treinado somente com uma amostra, e uma previsão será feita para ela. A acurácia, ou outra métrica de desempenho, pode então ser construída com o uso dessas previsões.

Outras preocupações relacionadas à criação de um conjunto de teste, como a seleção de um conjunto de teste out-of-time para problemas em que as observações do passado sejam usadas na previsão de eventos futuros, também são aplicáveis à validação cruzada.

No exercício 17, vimos que o ajuste de uma regressão logística com nossos dados de treinamento levou ao overfitting. Na verdade, o resultado do teste ($ROC\ AUC = 0.81$) foi substancialmente mais baixo que o do treinamento ($ROC\ AUC = 0.94$). Usamos pouca ou nenhuma regularização ao configurar o parâmetro C com um valor relativamente alto (1,000). Agora veremos o que acontece quando diversificamos C usando um amplo intervalo de valores.

Exercício 18: Reduzindo o overfitting no problema de classificação de dados sintéticos

Este exercício é a continuação do *Exercício 17: Gerando e modelando dados de classificação sintéticos*. Aqui, usaremos o procedimento de validação cruzada para encontrar um bom valor para o hiperparâmetro C . Faremos isso empregando apenas os dados de treinamento e reservando os dados de teste para depois que a construção do modelo for concluída. Prepare-se porque este será um longo exercício – mas mostraremos um procedimento geral que você poderá usar com vários tipos de modelos de machine learning, logo, o tempo gasto valerá a pena. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado em <http://bit.ly/2ZAY2Pr>.

1. Altere o valor do parâmetro de regularização, C , para que ele varie de $C = 1000$ a $C = 0.001$. Você pode usar os fragmentos de código a seguir para fazê-lo.

Primeiro, defina expoentes, que serão potências de 10, como mostrado em:

```
C_val_exponents = np.linspace(3,-3,13)
C_val_exponents
```

Aqui está a saída do código anterior:

```
array([ 3. , 2.5, 2. , 1.5, 1. , 0.5, 0. , -0.5, -1. , -1.5, -2. ,  
       -2.5, -3. ])
```

Agora varie C de acordo com as potências de 10, desta forma:

```
C_vals = np.float(10)**C_val_exponents  
C_vals
```

Esta é a saída do código anterior:

```
array([[1.00000000e+03, 3.16227766e+02, 1.00000000e+02, 3.16227766e+01,  
       1.00000000e+01, 3.16227766e+00, 1.00000000e+00, 3.16227766e-01,  
       1.00000000e-01, 3.16227766e-02, 1.00000000e-02, 3.16227766e-03,  
       1.00000000e-03])
```

Geralmente é uma boa ideia variar o parâmetro de regularização de acordo com potências de 10, ou usando uma estratégia semelhante, já que treinar modelos pode demorar muito, principalmente com a utilização da validação cruzada k-folds. Isso lhe dará uma boa noção de como um amplo intervalo de valores de C impacta o trade-off entre viés e variância, sem ser preciso treinar um número muito grande de modelos. Além das potências inteiras de 10, também incluímos pontos na escala de $\log_{10}$ que aparecem no caminho intermediário.

2. Importe a classe `roc_curve`:

```
from sklearn.metrics import roc_curve
```

Continuaremos a usar o resultado de ROC AUC para avaliar, treinar e testar o desempenho. Agora que temos vários valores de C para testar e vários folds (nesse caso quatro) para a validação cruzada, queremos armazenar os resultados de treinamento e teste de cada fold e de cada valor de C .

3. Defina uma função que receba o divisor da validação cruzada (`k_folds`), o array de valores de C (`C_vals`), o objeto de modelo (`model`), e as características e a variável de resposta (X e Y , respectivamente) como entradas, com os quais exploraremos diferentes níveis de regularização com a validação cruzada k-folds, usando o código a seguir:

```
def cross_val_C_search(k_folds, C_vals, model, X, Y):
```

Nota: A função que começamos nessa etapa retornará as ROC AUCs e os dados da curva ROC. O bloco de retorno será escrito durante uma etapa posterior do exercício. Por enquanto, você pode simplesmente escrever o código anterior como se encontra, porque definiremos `k_folds`, `C_vals`, `model`, `X` e `Y` à medida que avançarmos.

4. Crie um array NumPy para armazenar `C_vals`, com as dimensões `n_folds` por `len(C_vals)`:

```
n_folds = k_folds.n_splits
cv_train_roc_auc = np.empty((n_folds, len(C_vals)))
cv_test_roc_auc = np.empty((n_folds, len(C_vals)))
```

Em seguida, armazenaremos os arrays de limites e taxas de verdadeiros e falsos positivos que acompanham cada resultado da ROC AUC de teste em uma **lista de listas**.

Nota: Essa é uma maneira conveniente de armazenar todas as informações, já que uma lista em Python pode conter qualquer tipo de dado, inclusive outra lista. Aqui, cada item da **lista de listas** será uma tupla contendo os arrays de TPR, FPR e os limites de cada fold para cada valor de `C`. Isso deve ficar mais claro quando acessarmos esses arrays posteriormente para examiná-los.

5. Crie uma lista de listas vazias usando `[]` e `*len(C_vals)`:

```
cv_test_roc = [[]]*len(C_vals)
```

Usar `*len(C_vals)` implica que deve haver uma lista de tuplas com métricas (TPR, FPR, limites) para cada valor de `C`.

Aprendemos como percorrer os diferentes folds para executar a validação cruzada na seção anterior. O que temos de fazer agora é criar um loop externo, no qual aninharemos o loop da validação cruzada.

6. Crie um loop externo para treinar e testar cada um dos `k` folds para cada valor de `C`:

```
for c_val_counter in range(len(C_vals)):
    #Define o valor de C para o objeto de modelo
    model.C = C_vals[c_val_counter]
    #Conta os folds para cada valor de C
    fold_counter = 0
```

Podemos reutilizar o objeto de modelo que já temos e simplesmente definir um novo valor de `C` a cada execução do loop. Dentro do loop de valores de `C`, executaremos o loop de validação cruzada. Começaremos gerando os índices de linhas de dados de treinamento e teste para cada divisão.

7. Obtenha os índices de treinamento e teste de cada fold:

```
for train_index, test_index in k_folds.split(X, Y):
```

8. Indexe as características e a variável de resposta para obter os dados de treinamento e teste desse fold usando o código a seguir:

```
X_cv_train, X_cv_test = X[train_index], X[test_index]  
y_cv_train, y_cv_test = Y[train_index], Y[test_index]
```

Agora os dados de treinamento do fold atual serão usados para treinar o modelo.

9. Ajuste o modelo com os dados de treinamento, como mostrado em:

```
model.fit(X_cv_train, y_cv_train)
```

Isso “redefinirá” o modelo a partir de quaisquer que forem os coeficientes e a interceptação anteriores para refletir o treinamento com os dados novos.

Os resultados da ROC AUC de treinamento e teste serão então obtidos, assim como os arrays de TPRs, FPRs e limites que acompanham os dados de teste.

10. Obtenha o resultado da ROC AUC de treinamento:

```
y_cv_train_predict_proba = model.predict_proba(X_cv_train)  
cv_train_roc_auc[fold_counter, c_val_counter] = \  
roc_auc_score(y_cv_train, y_cv_train_predict_proba[:,1])
```

11. Obtenha o resultado da ROC AUC de teste:

```
y_cv_test_predict_proba = model.predict_proba(X_cv_test)  
cv_test_roc_auc[fold_counter, c_val_counter] = \  
roc_auc_score(y_cv_test, y_cv_test_predict_proba[:,1])
```

12. Obtenha as curvas ROC de teste para cada fold usando o código a seguir:

```
this_fold_roc = roc_curve(y_cv_test, y_cv_test_predict_proba[:,1])  
cv_test_roc[c_val_counter].append(this_fold_roc)
```

Usaremos um contador para registrar o incremento dos folds, e, quando estivermos fora do loop de validação cruzada, exibiremos uma atualização de status na saída padrão. Ao executar procedimentos de processamento demorado, é uma boa ideia exibir periodicamente o status da tarefa para que você possa monitorar seu progresso e confirmar se tudo continua funcionando corretamente. Provavelmente esse procedimento de validação cruzada só levará alguns segundos em seu laptop, mas para tarefas mais longas isso pode ser tranquilizador.

13. Incremente o contador de folds usando este código:

```
fold_counter += 1
```

14. Escreva o código a seguir para mostrar o progresso da execução de cada valor de C:

```
print('Done with C = {}'.format(lr_syn.C))
```

15. Escreva o código que retornará as ROC AUCS e os dados da curva ROC e encerrará a função:

```
return cv_train_roc_auc, cv_test_roc_auc, cv_test_roc
```

Continuaremos usando a divisão em três folds que mostramos anteriormente, mas recomendo que você teste esse procedimento com diferentes números de folds para comparar o efeito.

Abordamos muito material nas etapas anteriores. Se quiser, reserve algum tempo para fazer uma revisão com seus colegas de sala e se certificar se entendeu cada parte. Executar a função é comparativamente fácil. É aí que reside a beleza de uma função bem projetada – todas as partes complicadas ficam isoladas, permitindo-lhe se concentrar no uso.

16. Execute a função para procurar os valores de C que definimos, usando o modelo e os dados com os quais trabalhamos no exercício anterior e o código a seguir:

```
cv_train_roc_auc, cv_test_roc_auc, cv_test_roc = \
cross_val_C_search(n_folds, C_vals, lr_syn, X_syn_train, y_syn_train)
```

Quando você executar esse código, deve ver a seguinte saída aparecer abaixo da célula de código à medida que a validação cruzada terminar para cada valor de C .

```
Done with C = 1000.0
Done with C = 316.22776601683796
Done with C = 100.0
Done with C = 31.622776601683793
Done with C = 10.0
Done with C = 3.1622776601683795
Done with C = 1.0
Done with C = 0.31622776601683794
Done with C = 0.1
Done with C = 0.03162277660168379
Done with C = 0.01
Done with C = 0.0031622776601683794
Done with C = 0.001
```

Qual será a aparência dos resultados da validação cruzada? Há algumas maneiras de examinarmos isso. É útil verificar o desempenho de cada fold individualmente para que você veja como os resultados variam.

Isso mostrará como os diferentes subconjuntos de dados se saem como conjuntos de teste, levando a uma ideia geral do nível de desempenho que pode ser esperado a partir do conjunto de teste desconhecido. Aqui queremos saber se podemos ou não usar a

regularização para reduzir o overfitting que vimos. Sabemos que usar $C = 1,000$ leva ao overfitting – sabemos disso porque comparamos os resultados de treinamento e teste. Porém, e quanto aos outros valores de C que testamos? Uma boa maneira de visualizar seria plotar os resultados de treinamento e teste no *eixo y* e os valores de C no *eixo x*.

17. Percorra cada fold para visualizar seus resultados individualmente usando o código a seguir:

for this_fold in range(n_folds):

```
plt.plot(C_val_exponents, cv_train_roc_auc[this_fold], '-o',  
        color = cmap(this_fold), label = "Training fold {}".format(this_fold + 1))  
plt.plot(C_val_exponents, cv_test_roc_auc[this_fold], '-x',  
        color = cmap(this_fold), label = "Testing fold {}".format(this_fold + 1))
```

```
plt.ylabel('ROC AUC')
```

```
plt.xlabel('log$_{10}$$(C)$')
```

```
plt.legend(loc = [1.1, 0.2])
```

```
plt.title('Cross validation scores for each fold')
```

Você obterá a seguinte saída:

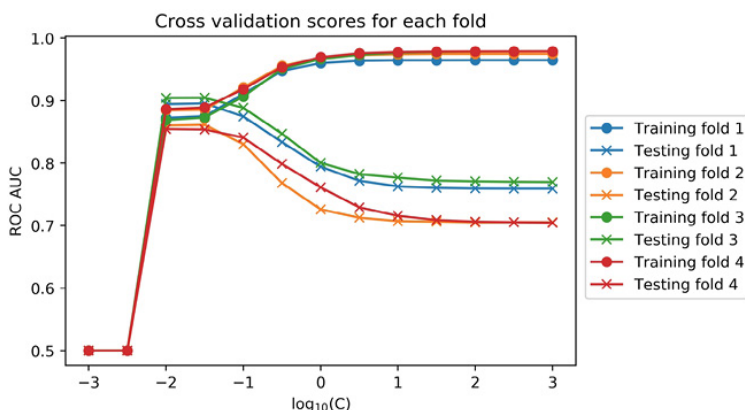


Figura 4.20 – Resultados de treinamento e teste para cada fold e valor de C .

Podemos ver que, à medida que C diminui para cada fold da validação cruzada, o desempenho do treinamento também diminui. No entanto, ao mesmo tempo o desempenho do teste aumenta. Para alguns folds e valores de C , na verdade o resultado da ROC AUC de teste excede o dos dados de treinamento, enquanto, para outros, essas duas métricas simplesmente se aproximam. Seja como for, podemos dizer que os valores de C iguais a $10^{-1.5}$ e 10^{-2} têm um desempenho de teste semelhante, que é substancialmente mais alto que o de $C = 10^3$. Logo, parece que a regularização resolveu com sucesso nosso problema de overfitting.

No entanto, e quanto aos valores mais baixos de C ? Para valores menores que $10^{-1.5}$, a métrica ROC AUC cai repentinamente para 0.5. Como você sabe, esse valor significa que o modelo de classificação é basicamente inútil, com um desempenho que não é melhor que o lançamento de uma moeda. Recomendo que você verifique isso posteriormente ao examinar como a regularização afeta os valores dos coeficientes; de qualquer forma, é isso que acontece quando é aplicada uma regularização L1 em um nível tão alto a ponto de todos os coeficientes do modelo serem reduzidos a 0. É claro que esses modelos não são úteis para nós.

É útil examinar o desempenho de treinamento e teste de cada divisão da validação k -folds para conhecermos a variabilidade esperada no desempenho do modelo quando ele é avaliado com dados novos desconhecidos. Porém, para resumir os resultados do procedimento k -folds, uma abordagem comum é calcular a média da métrica de desempenho dos folds para cada valor do hiperparâmetro que está sendo considerado. Faremos isso na próxima etapa.

18. Plote a média de resultados da ROC AUC de treinamento e teste para cada valor de C usando o código a seguir:

```
plt.plot(C_val_exponents, np.mean(cv_train_roc_auc, axis=0), '-o',  
        label='Average training score')  
plt.plot(C_val_exponents, np.mean(cv_test_roc_auc, axis=0), '-x',  
        label='Average testing score')  
plt.ylabel('ROC AUC')  
plt.xlabel('log10$(C)$')  
plt.legend()  
plt.title('Cross validation scores averaged over all folds')
```

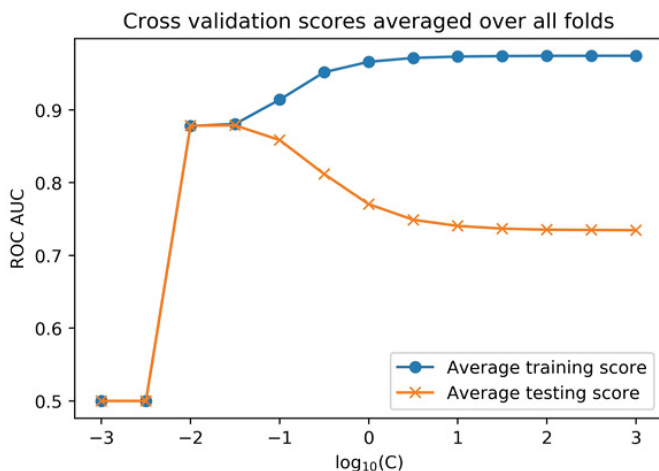


Figura 4.21 – Resultados da média de treinamento e teste nos folds da validação.

Nessa plotagem, fica claro que $C = 10^{-1.5}$ e 10^{-2} são os melhores valores para C . Há pouco ou nenhum overfitting aqui, já que os resultados da média de treinamento e teste são quase iguais. Você poderia pesquisar em uma grade de valores mais refinados (isto é, $C = 10^{-1.1}$, $10^{-1.2}$ e assim por diante) para localizar com maior precisão um valor de C . No entanto, em nosso gráfico podemos ver que provavelmente $C = 10^{-1.5}$ ou $C = 10^{-2}$ serão boas soluções. Daremos continuidade com $C = 10^{-1.5}$.

Examinar a síntese da métrica ROC AUC é uma boa maneira de termos uma ideia rápida de como os modelos se sairão. Porém, para qualquer aplicação empresarial do mundo real, com frequência temos de selecionar um limite específico que acompanhe as taxas de verdadeiros e falsos positivos. Isso é necessário para usarmos o classificador e tomar a obrigatória decisão “sim” ou “não”, que no nosso estudo de caso é uma previsão de se uma conta ficará ou não inadimplente. Portanto, é útil examinar as curvas ROC dos diferentes folds da validação cruzada. Para facilitar isso, a função anterior foi projetada para retornar as taxas de verdadeiros e falsos positivos, e os limites, de cada fold de teste e valor de C , na lista de listas `cv_test_roc`. Primeiro, temos de encontrar o índice da lista externa que corresponde ao valor de C que selecionamos, $10^{-1.5}$.

Para encontrá-lo, poderíamos apenas examinar nossa lista de valores de C e fazer a contagem manualmente, porém é mais seguro fazê-lo programaticamente encontrando o índice do elemento diferente de zero de um array booleano como mostrado na próxima etapa.

19. Use um array booleano para encontrar um índice em que $C = 10^{-1.5}$ e converta-o para um tipo de dado inteiro com este código:

```
best_C_val_bool = C_val_exponents == -1.5
best_C_val_bool.astype(int)
```

Aqui está a saída do código anterior:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0])
```

20. Converta a versão de tipo inteiro do array booleano em um único índice inteiro usando a função `nonzero` com o código a seguir:

```
best_C_val_ix = np.nonzero(best_C_val_bool.astype(int)) best_C_val_ix[0][0]
best_C_val_ix[0][0]
```

Esta é a saída do código anterior:

Localizamos com sucesso o valor de C que queremos usar.

21. Acesse as taxas de verdadeiros e falsos positivos para plotar as curvas ROC de cada fold:

```
for this_fold in range(n_folds):  
    fpr = cv_test_roc[best_C_val_ix[0][0]][this_fold][0]  
    tpr = cv_test_roc[best_C_val_ix[0][0]][this_fold][1]  
    plt.plot(fpr, tpr, label='Fold {}'.format(this_fold + 1))  
plt.xlabel('False positive rate')  
plt.ylabel('True positive rate')  
plt.title('ROC curves for each fold at  $C = 10^{-1.5}$ ')  
plt.legend()
```

A saída pode ser visualizada na Figura 4.22.

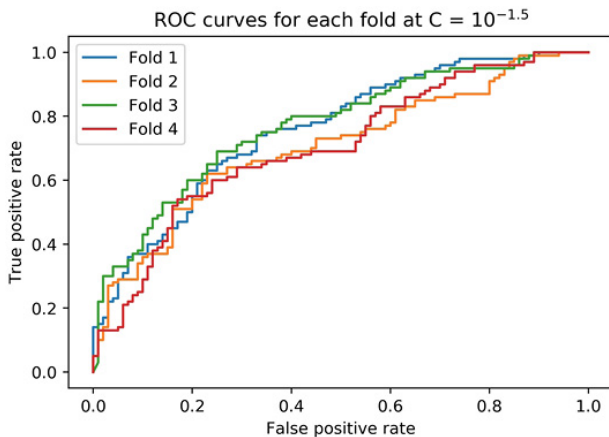


Figura 4.22 – Curvas ROC de cada fold.

Parece que há bastante variabilidade nas curvas ROC. Por exemplo, se por alguma razão limitarmos a taxa de falsos positivos a 40%, então, pela plotagem parece que talvez possamos atingir uma taxa de verdadeiros positivos de aproximadamente 65% a 80%. Você pode encontrar os valores exatos examinando os arrays que plotamos. Isso lhe dará uma ideia de quanta variabilidade pode ser esperada do desempenho na implantação do modelo com novos dados. Geralmente, quanto maior o número de dados de treinamento disponíveis, menor é a variabilidade entre os folds da validação cruzada, logo, esse também pode ser um sinal de que seria bom coletar dados adicionais, principalmente se a variabilidade entre os folds de treinamento parecer alta. Se quiser, faça testes também com diferentes números de folds usando esse procedimento para ver o efeito sobre a variabilidade de resultados entre os folds.

Normalmente faríamos testes com outros modelos em nosso

problema de dados sintéticos, como uma floresta aleatória ou uma máquina de vetores de suporte, mas como na validação cruzada a regressão logística provou ser o melhor modelo, vamos torná-la nossa opção final. Quando o modelo final tiver sido selecionado, todos os dados de treinamento poderão ser usados no seu ajuste com os hiperparâmetros escolhidos com a validação cruzada. É recomendável usar o maior número de dados possível no ajuste do modelo, já que geralmente os modelos funcionam melhor quando treinados com mais dados.

22. Treine a regressão logística com todos os dados de treinamento de nosso problema sintético e compare os resultados de treinamento e teste, usando o conjunto reservado para teste como mostrado nas etapas a seguir.

Nota: Esta é a última etapa do processo de seleção do modelo. Você só deve usar o conjunto de teste desconhecido após sua busca do modelo e dos hiperparâmetros terminar, caso contrário ele não será “desconhecido”.

23. Defina o valor de C e treine o modelo com todos os dados de treinamento usando este código:

```
lr_syn.C = 10**(-1.5)
lr_syn.fit(X_syn_train, y_syn_train)
```

Aqui está a saída do código anterior:

```
LogisticRegression(C=0.03162277660168379, class_weight=None, dual=False,
    fit_intercept=True, intercept_scaling=1, max_iter=100,
    multi_class='warn', n_jobs=None, penalty='l1', random_state=1,
    solver='liblinear', tol=0.0001, verbose=0, warm_start=False)
```

24. Obtenha as probabilidades previstas e o resultado da ROC AUC para os dados de treinamento com este código:

```
y_syn_train_predict_proba = lr_syn.predict_proba(X_syn_train)
roc_auc_score(y_syn_train, y_syn_train_predict_proba[:,1])
```

Esta é a saída:

```
0.8802812499999999
```

25. Obtenha as probabilidades previstas e o resultado da ROC AUC para os dados de teste com este código:

```
y_syn_test_predict_proba = lr_syn.predict_proba(X_syn_test)
roc_auc_score(y_syn_test, y_syn_test_predict_proba[:,1])
```

A saída do código anterior é:

```
0.8847884788478848
```


Podemos ver que, com o uso da regularização, os resultados de treinamento e teste do modelo são semelhantes, indicando que o problema do overfitting foi resolvido. O resultado do treinamento é mais baixo, já que introduzimos viés no modelo para compensar a variância. No entanto, não há problema, já que o resultado do teste, que é a parte mais importante, é mais alto. O resultado do teste fora da amostra é o que importa para a capacidade preditiva. Recomendo que você verifique se esses resultados de treinamento e teste são semelhantes aos do procedimento de validação cruzada exibindo os valores dos arrays que plotamos posteriormente; verá que eles são semelhantes.

Nota: No mundo real, antes de distribuir esse modelo para seu cliente para uso na produção, provavelmente você o treinaria com todos os dados que recebeu, inclusive o conjunto de teste desconhecido. Esse comportamento se baseia novamente na ideia de que, quanto maior o número de dados que um modelo vê, melhor deve ser seu desempenho na prática.

Sabemos que a regularização L1 funciona diminuindo a magnitude (isto é, o valor absoluto) dos coeficientes da regressão logística. Ela também configura alguns coeficientes com zero, executando assim a seleção de características. Na próxima etapa, determinaremos quantos coeficientes foram configurados com zero.

26. Acesse os coeficientes do modelo treinado e determine quantos não são iguais a zero ($\neq 0$) com este código:

```
sum((lr_syn.coef_ != 0)[0])
```

A saída deve ser: 2

O código usa a soma de um array booleano para indicar onde ficam localizados os coeficientes diferentes de zero, logo, ele exibe quantos coeficientes do modelo não foram configurados com zero pela regularização L1. Só 2 das 200 características foram selecionadas!

27. Examine o valor da interceptação usando o código a seguir:

```
lr_syn.intercept_
```

A saída deve ser:

```
array([0.])
```

Ela mostra que a interceptação foi regularizada com 0.

Neste exercício, atingimos vários objetivos. Usamos o procedimento de validação cruzada k-folds para ajustar o hiperparâmetro de

regularização. Vimos o poder da regularização na redução do overfitting, e, no caso da regularização L1 na regressão logística, a seleção de características.

Muitos algoritmos de machine learning oferecem algum tipo de recurso de seleção de características. Vários também requerem o ajuste dos hiperparâmetros. A função exibida aqui para percorrer os hiperparâmetros, e executar a validação cruzada, é um conceito poderoso que pode ser generalizado para outros modelos. O scikit-learn oferece uma funcionalidade que facilita esse processo; estamos falando do procedimento `sklearn.model_selection.GridSearchCV`, que aplica a validação cruzada à busca de hiperparâmetros em uma grade. A **busca na grade** pode ser útil quando houver vários hiperparâmetros a serem ajustados, com a verificação de todas as combinações dos intervalos de diferentes hiperparâmetros especificados. Uma **busca em grade randomizada** pode acelerar esse processo selecionando aleatoriamente um número menor de combinações se uma exaustiva busca na grade demorar muito. Quando você se sentir confortável com os conceitos ilustrados aqui, recomendo que otimize seu fluxo de trabalho com funções convenientes como essas.

Opções da regressão logística no scikit-learn

Usamos e discutimos a maioria das opções que você pode fornecer para o scikit-learn ao instanciar ou ajustar os hiperparâmetros de uma classe de modelo `LogisticRegression`. Aqui, listaremos todas e daremos alguns conselhos gerais sobre seu uso:

Notas e recomendações para a seleção	
Regularização	de coeficientes L1 (lasso) ou L2 (ridge). O L1 executa a seleção de características e o L2 não. O modelo de melhor desempenho geral deve ser avaliado com o uso de ambos.
Grid	Relacionado ao algoritmo de otimização usado para encontrar coeficientes. A documentação diz "implementado apenas para penalização l2 com solver liblinear. Use <code> dual = False</code> quando <code> n_samples > n_features</code> ".
max_iter	o número de iterações da alteração de valores até o algoritmo de otimização ser interrompido. Essa é a única maneira de controlar por quanto tempo a otimização será executada e quão próxima do valor ideal será a solução.
penalty	parâmetro da regularização L1 ou L2. Precisa ser determinado com o uso de um conjunto de validação, ou validação cruzada.
intercept_scaling	um termo de interceptação deve ser estimado. A menos que você tenha certeza de que não vai precisar de uma interceptação, é melhor ter uma.
fit_intercept	se usar para impedir a regularização da interceptação, quando usado o solver liblinear.
class_weight	se o significado da distribuição é importante para a classificação, é melhor usar o processo de treinamento. Caso contrário, todas as amostras serão consideradas "igualmente importantes" no ajuste do modelo. Pode ser útil para datasets desbalanceados: nesse caso, tente usar "balanced".

Solver	parâmetro gerador de números aleatórios usada por certos algoritmos de solver.
Solver	um novo tipo de algoritmo (saga) usado para estimar os parâmetros do modelo. Consulte a discussão do capítulo anterior ou a documentação sobre as vantagens e desvantagens dos diferentes solvers.
max_iter	número máximo de iterações do algoritmo, que controla o quão próxima dos parâmetros ideais está a solução. Se você se deparar com um aviso de que o algoritmo de solução não convergiu, pode tentar aumentar esse número.
MultiClass	ótima para a classificação multiclasse; não faz parte do escopo deste livro.
verbose	a natureza da saída no terminal durante o procedimento de otimização.
WarmStart	o objeto de modelo foi reutilizado em vários treinamentos, indica se usaremos a solução anterior como ponto de partida do próximo procedimento de otimização.
MultiThreaded	modo processadores usados no processamento paralelo, no caso da classificação multiclasse "ovr".

Figura 4.23 – Lista completa de opções para o modelo de regressão logística no scikit-learn.

Se você tiver dúvidas com relação a que opção usar para a regressão logística, recomendamos que consulte a documentação do scikit-learn para ter uma orientação melhor (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression). Algumas opções, como o parâmetro de regularização C, ou a escolha da regularização L1 ou L2, têm de ser examinadas por meio do processo de validação cruzada. Aqui, como em várias decisões que devem ser tomadas na ciência de dados, não há uma abordagem universal aplicável a todos os datasets. A melhor maneira de saber que opções usar com um dataset específico é testar várias delas e ver qual fornece o melhor desempenho fora da amostra. A validação cruzada oferece um modo robusto de fazer isso.

Escalonamento de dados, pipelines e características de interação no scikit-learn

Escalonamento de dados

Em comparação com os dados sintéticos com os quais acabamos de trabalhar, os dados do estudo de caso são relativamente maiores. Se quisermos usar a regularização L1, então, de acordo com a documentação oficial (https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression), temos de utilizar o solver saga. No entanto, esse solver não é robusto para datasets não escalonados. Logo, precisamos nos certificar de escalonar os dados. Essa também é uma boa ideia sempre que executarmos a regularização para que todas as características estejam na mesma escala e sejam igualmente penalizadas pelo processo de regularização. Uma maneira simples de assegurar que todas as características tenham a mesma escala é fazê-las passar pela transformação de diminuição do valor mínimo e

divisão pelo intervalo do valor mínimo ao valor máximo. Isso transformará cada característica para que ela tenha um mínimo igual a 0 e um máximo igual a 1. Para instanciar o escalonador `MinMaxScaler` que executa esse processo, podemos usar o código a seguir:

```
from sklearn.preprocessing import MinMaxScaler
min_max_sc = MinMaxScaler()
```

Pipelines

Anteriormente, usamos um modelo de regressão logística no loop de validação cruzada. No entanto, agora que estamos escalonando os dados, que considerações novas devemos examinar? O escalonamento é “aprendido” a partir dos valores mínimo e máximo dos dados de treinamento. Depois o modelo de regressão logística seria treinado com dados escalonados pelas extremidades dos dados de treinamento. Porém, não sabemos os valores mínimo e máximo dos dados novos desconhecidos. Logo, seguindo a filosofia de tornar a validação cruzada um indicador eficaz do desempenho fora da amostra, temos de usar os valores mínimo e máximo dos dados de treinamento em cada fold da validação cruzada para escalonar os dados de teste desse fold antes de fazer previsões com eles. O scikit-learn otimizou esse processo para nós na forma de uma única ferramenta: o Pipeline. Nosso pipeline será composto de duas etapas: o **escalonador** e o **modelo de regressão logística**. Elas podem ser ajustadas nos dados de treinamento e depois usadas para previsões com os dados de teste. Essas etapas ocorrem simultaneamente, como se o pipeline fosse uma única entidade. Veja como um Pipeline é instanciado:

```
from sklearn.pipeline import Pipeline
scale_lr_pipeline = Pipeline(steps=[('scaler', min_max_sc), ('model', lr)])
```

Características de interação

Considerando os dados do estudo de caso, qual foi o resultado da discussão de se um modelo de regressão logística com todas as características possíveis sofreria overfitting ou underfitting? Você pode pensar nisso da perspectiva de regras práticas como a “regra de 10” e do número de características (17) versus as amostras (26.664) que temos. Alternativamente, pode considerar todo o trabalho que fizemos até agora com esses dados. Por exemplo, pudemos visualizar todas as características e verificar se fazem sentido. Já que há relativamente poucas características, e temos uma confiança

razoavelmente alta de que elas têm alta qualidade devido ao nosso trabalho de exploração de dados, estamos em uma situação diferente da dos exercícios de dados sintéticos deste capítulo, nos quais tínhamos muitas características sobre as quais sabíamos pouco. Logo, pode ser que o overfitting seja uma questão menor no problema de nosso estudo de caso nesse momento, e os benefícios da regularização não serão significativos.

Na verdade, podemos causar underfitting no modelo usando somente as 17 características que vieram com os dados. Uma estratégia que resolve isso é a criação de novas características por engenharia. Um método de engenharia de características simples que discutimos é usar características de interação (interaction features) ou polinomiais. Polinômios podem não fazer sentido dada a maneira como alguns dos dados foram codificados; por exemplo, $-1^2 = 1$, que pode não ser sensato para PAY_1. No entanto, podemos tentar criar características de interação para capturar os relacionamentos entre as características. PolynomialFeatures pode ser usado para criar somente características de interação, sem características polinomiais. Um exemplo de código seria:

```
make_interactions = PolynomialFeatures(degree=2, interaction_only=True,
                                       include_bias=False)
```

Aqui, degree representa o grau das características polinomiais, interaction_only usa um valor booleano (configurá-lo com True indica que só características de interação serão criadas), assim como include_bias, que adiciona uma interceptação ao modelo (o valor padrão é False, que é o correto nesse caso, já que o modelo de regressão logística adicionará uma interceptação).

Atividade 4: Validação cruzada e engenharia de características com os dados do estudo de caso

Nesta atividade, aplicaremos o conhecimento de validação cruzada e regularização que obtivemos neste capítulo aos dados do estudo de caso. Executaremos uma engenharia de características básica. Para estimar parâmetros para o modelo de regressão logística regularizado dos dados do estudo de caso, que têm tamanho maior do que os dados sintéticos com os quais trabalhamos, usaremos o solver saga. Para utilizar esse solver, e para fins de regularização, teremos de **escalonar** nossos dados como parte do processo de modelagem, o que nos levará a usar os Pipelines do scikit-learn. Quando você terminar a atividade, terá obtido um desempenho de teste de validação cruzada melhor com

o uso de características de interação, como mostrado na Figura 4.24:

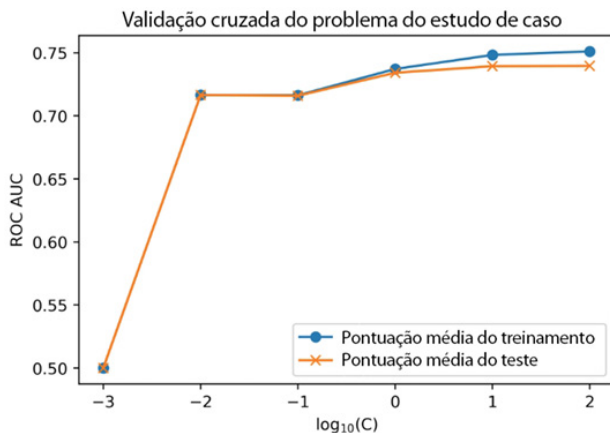


Figura 4.24 – Desempenho de teste do modelo melhorado.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e a saída resultante desta atividade foram carregados em um Jupyter Notebook e podem ser encontrados em <http://bit.ly/2Z53aX4>.

1. Selecione as características no DataFrame dos dados do estudo de caso.

Você pode usar a lista de nomes de características que criamos neste capítulo. No entanto, certifique-se de não incluir a variável de resposta, que seria uma característica muito boa (mas totalmente inapropriada)!

2. Crie uma divisão de treinamento/teste usando um seed aleatório igual a 24.

Usaremos essa divisão ao avançar e reservaremos os dados de teste como o conjunto de teste desconhecido. Dessa forma, poderemos criar facilmente notebooks separados com outras abordagens de modelagem, usando os mesmos dados de treinamento.

3. Instancie `MinMaxScaler` para escalonar os dados.
4. Instancie um modelo de regressão logística com o solver `saga` e a penalização `L1`, e configure `max_iter` com 1.000 já que queremos que o solver tenha iterações suficientes para encontrar uma boa solução.
5. Importe a classe `Pipeline` e crie um `Pipeline` com o escalonador e o modelo de regressão logística, usando os nomes `'scaler'` e `'model'` para as etapas, respectivamente.

6. Use os métodos `get_params` e `set_params` para visualizar os parâmetros de cada estágio do pipeline e alterá-los.
7. Crie um intervalo menor de valores de C para testar com a validação cruzada, já que esses modelos demandarão mais tempo para serem treinados e testados com mais dados do que nossos exercícios anteriores; recomendamos $C = [10^2, 10, 1, 10^{-1}, 10^{-2}, 10^{-3}]$.
8. Crie uma nova versão da função `cross_val_C_search`, chamada `cross_val_C_search_pipe`. Em vez do argumento `model`, essa função receberá um argumento `pipeline`. As alterações na função serão a definição do valor de C com o uso de `set_params(model_C = <valor que você quiser testar>)` no pipeline, a substituição do modelo pelo pipeline para os métodos `fit` e `predict_proba` e o acesso ao valor de C com o uso de `pipeline.get_params()['model_C']` para a atualização do status exibido.
9. Execute essa função como no exercício anterior, mas usando o novo intervalo de valores de C , o pipeline que você criou e as características e a variável de resposta da divisão de treinamento dos dados do estudo de caso.

Você pode ver avisos aqui, ou em etapas posteriores, sobre a não convergência do solver; poderíamos tentar usar as opções `tol` ou `max_iter` para chegar à convergência, mas os resultados obtidos com `max_iter = 1000` devem ser suficientes.
10. Plote a ROC AUC da média de treinamento e teste entre os folds para cada valor de C .
11. Crie características de interação para os dados do estudo de caso e confirme se o número de características novas faz sentido.
12. Repita o procedimento de validação cruzada e observe o desempenho do modelo agora.

Levaremos mais tempo devido ao número maior de características, mas provavelmente serão apenas alguns minutos. Você acha que o desempenho médio do teste de validação cruzada melhorou com as características de interação? A regularização foi útil?

Nota: A solução para esta atividade consta na página 303.

Resumo

Neste capítulo, introduzimos os últimos detalhes da regressão logística

e continuamos usando o scikit-learn para ajustar modelos. Visualizamos melhor como o processo de ajuste do modelo funciona conhecendo o conceito de função custo, que é reduzida pelo procedimento de gradiente decrescente na estimativa de parâmetros do modelo durante o ajuste.

Também examinamos a necessidade de regularização, introduzindo os conceitos de underfitting e overfitting. Para reduzir o overfitting, vimos como ajustar a função custo para regularizar os coeficientes de um modelo de regressão logística usando a penalização L1 ou L2. Empregamos a validação cruzada para selecionar o nível de regularização, ajustando o hiperparâmetro. Para reduzir o underfitting, testamos uma técnica de engenharia de características simples criando características de interação para os dados do estudo de caso.

Agora estamos familiarizados com alguns dos conceitos mais importantes de machine learning. Até aqui, só usamos um modelo de classificação muito básico, isto é, a regressão logística. No entanto, mesmo aprendendo a usar mais modelos, veremos que os conceitos de overfitting, underfitting, trade-off entre viés e variância, e ajuste de hiperparâmetros surgirão de tempos em tempos. Essas ideias, assim como as convenientes implementações do scikit-learn para as funções de validação cruzada que usei neste capítulo, nos ajudarão em nossa exploração de métodos de previsão mais avançados.

No próximo capítulo, conheceremos as árvores de decisão, um tipo de modelo preditivo muito diferente da regressão logística, e as florestas aleatórias que se baseiam nelas. Porém, usaremos os mesmos conceitos de validação cruzada e busca de hiperparâmetros que aprendemos aqui, para ajustar esses modelos.

Árvores de decisão e florestas aleatórias

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Treinar um modelo de árvore de decisão no scikit-learn
- Usar o Graphviz para visualizar um modelo de árvore de decisão treinado
- Formular as funções custo usadas para dividir nós em uma árvore de decisão
- Executar uma busca de hiperparâmetro em grade usando a validação cruzada com funções do scikit-learn
- Treinar um modelo de floresta aleatória no scikit-learn
- Avaliar as características mais importantes em um modelo de floresta aleatória

Este capítulo introduzirá as árvores de decisão e as florestas aleatórias do scikit-learn, além de descrever o método para a execução da busca de hiperparâmetros em grade.

Introdução

Nos dois últimos capítulos, entendemos na totalidade o funcionamento da regressão logística. Também ganhamos muita experiência no uso do pacote scikit-learn de Python para criar modelos de regressão.

Neste capítulo, introduziremos um tipo de modelo preditivo poderoso que usa uma abordagem completamente diferente do modelo de regressão logística: as **árvores de decisão**. O conceito do uso de um processo em árvore para tomar decisões é simples e, portanto, os

modelos de árvore de decisão são fáceis de interpretar. Porém, uma crítica comum às árvores de decisão é a de que elas sobreajustam os dados de treinamento. Para remediar esse problema, os pesquisadores desenvolveram **combinações de métodos (ensemble methods)**, como as **florestas aleatórias**, que combinam muitas árvores de decisão para operar juntas e fazer previsões melhores do que qualquer árvore individual faria.

Veremos que as árvores de decisão e as florestas aleatórias podem melhorar a qualidade da modelagem preditiva dos dados do estudo de caso além do que conseguimos até agora com a regressão logística.

Árvores de decisão

As **árvores de decisão** e os modelos de machine learning que se baseiam nelas, em particular as **florestas aleatórias** e as **árvores de aumento de gradiente (gradient boosted trees)**, são tipos de modelos totalmente diferentes dos modelos lineares generalizados, como a regressão logística. Os GLMs têm sua origem nas teorias de estatística clássica, que têm uma longa história. Os conceitos matemáticos existentes por trás da regressão linear foram desenvolvidos originalmente no começo do século 19 por Legendre e Gauss. É por isso que a distribuição normal também é chamada de gaussiana.

Por outro lado, embora a ideia de usar um processo em árvore para tomar decisões seja relativamente simples, a popularidade das árvores de decisão como modelos matemáticos aumentou recentemente. Os procedimentos matemáticos que usamos atualmente para formular árvores de decisão no contexto da modelagem preditiva foram publicados nos anos 1980. A razão para esse desenvolvimento mais recente é que os métodos usados para o crescimento de árvores de decisão dependem de poder computacional – isto é, a habilidade de processar muitos números rapidamente. Hoje temos essa capacidade, mas ela só ficou amplamente disponível há pouco tempo na história da matemática.

Porém, o que é uma árvore de decisão? Podemos ilustrar o conceito básico usando um exemplo prático. Suponhamos que você estivesse considerando se deve ou não sair de casa em um dia específico. A única informação na qual baseará sua decisão envolve o clima e, em particular, se o sol está brilhando e se está muito quente. Se for um

dia ensolarado, sua tolerância a temperaturas baixas ficará mais alta e você sairá se estiver fazendo pelo menos 10 °C.

No entanto, se estiver nublado, você precisará de temperaturas mais quentes e só sairá se estiver fazendo 15 °C ou mais. Seu processo de tomada de decisão poderia ser representado pela árvore a seguir:

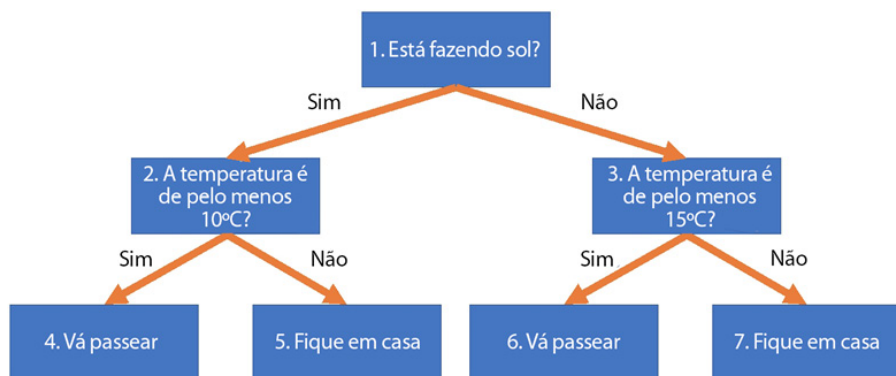


Figura 5.1 – Árvore de decisão de se é possível sair de acordo com o clima.

Como você pode ver, as árvores de decisão têm estrutura intuitiva e imitam a maneira como decisões lógicas são tomadas pelos humanos. Logo, são um tipo de modelo matemático altamente **interpretável**, o que pode ser uma propriedade desejável dependendo do público-alvo. Por exemplo, o cliente de um projeto de ciência de dados poderia querer entender claramente como um modelo funciona. As árvores de decisão são um bom modo de atender a esse requisito, contanto que seu desempenho seja suficiente.

Terminologia das árvores de decisão e conexões com o Machine Learning

Examinando a árvore da *Figura 5.1*, podemos começar a nos familiarizar com a terminologia das árvores de decisão. Já que há dois níveis de decisões sendo tomadas, com base nas condições de nebulosidade no primeiro nível e na temperatura no segundo nível, dizemos que essa árvore de decisão tem uma **profundidade** igual a dois. Os dois **nós** do segundo nível são decisões baseadas na temperatura, mas os tipos de decisões poderiam ser diferentes dentro de um nível; por exemplo, se quiséssemos basear nossa decisão em se está ou não chovendo, caso não fizesse sol.

No contexto do machine learning, os fatores que são usados para a tomada de decisões nos nós (em outras palavras, para **dividir** os nós)

são as características. No exemplo da *Figura 5.1*, temos características categóricas binárias para se está fazendo sol e uma característica contínua relativa à temperatura. Embora só tenhamos mostrado cada característica sendo usada uma única vez em um galho específico da árvore, a mesma característica poderia ser usada várias vezes em um galho. Por exemplo, se quiséssemos sair em um dia de sol em que fizesse pelo menos 10 °C, não sairíamos se estivesse mais de 40 °C – quente demais! Nesse caso, o nó 4 da *Figura 5.1* seria dividido na condição “A temperatura é de pelo menos 40 °C?”, para a qual o resultado seria “fique em casa “ se a resposta fosse “sim” e “vá passear” se a resposta fosse “não”, significando que a temperatura está entre 10 °C e 40 °C. Logo, as árvores de decisão conseguem capturar efeitos não lineares das características, ao contrário de um relacionamento linear que assumiria que, quanto mais quente estiver, mais estaremos propensos a sair.

Considere a maneira como as árvores costumam ser representadas, como na *Figura 5.1*. Os galhos crescem para baixo de acordo com as decisões binárias que podem causar a divisão em mais dois nós. Essas decisões binárias podem ser consideradas como regras “se, então”. Isto é, se certo critério for atendido, faça isso, caso contrário, faça aquilo. A decisão que está sendo tomada na árvore de nosso exemplo é análoga ao conceito de variável de resposta em machine learning. Se criássemos uma árvore de decisão para o problema do estudo de caso de não pagamento de dívida, as decisões seriam previsões dos valores de resposta binários, que são “essa conta está inadimplente” ou “essa conta não está inadimplente”. Uma árvore que responde ao tipo de pergunta binária sim/não é chamada de **árvore de classificação**. No entanto, as árvores de decisão são muito versáteis e também podem ser usadas para a classificação multiclasse e a regressão.

Os nós terminais da base da árvore são chamados de **folhas** ou nós folhas. Em nosso exemplo, as folhas são as decisões finais de se devemos sair ou ficar em casa. Há quatro folhas em nossa árvore, mas, se ela tivesse profundidade igual a um, e quiséssemos tomar a decisão com base apenas nas condições de nebulosidade, haveria duas folhas, e os nós 2 e 3 da *Figura 5.1* seriam nós folhas com “vá passear” e “fique em casa” como decisões, por exemplo.

No exemplo, cada nó de cada nível antes do nível final foi dividido. Isso não é estritamente necessário, já que você pode sair em um dia de sol independentemente da temperatura. Nesse caso, o nó 2 não será dividido, logo, esse galho da árvore terminará no primeiro nível com

uma decisão “sim”. Sua decisão para dias enevoados, entretanto, pode envolver a temperatura, significando que esse galho pode se estender por mais um nível. Contudo, no caso em que cada nó é dividido antes do último nível, pense na rapidez com que o número de folhas pode crescer com o número de níveis.

Imagine o que aconteceria se estendêssemos a árvore de decisão da *Figura 5.1* para baixo com um nível adicional, talvez com uma característica de velocidade do vento, para incluir o vento frio nas quatro combinações de condições de nebulosidade e temperatura. Cada um dos quatro nós que agora são folhas (os nós numerados de quatro a sete na *Figura 5.1*) seriam divididos em mais dois nós folhas, considerando-se a velocidade do vento em cada caso. Então, haveria $4 \times 2 = 8$ nós folhas. É claro que em uma árvore com n níveis, em que cada nó antes do último nível é dividido, haverá 2^n nós folhas. É importante lembrar-se disso, já que a **profundidade máxima** é um dos hiperparâmetros que você pode definir para um classificador de árvore de decisão no scikit-learn. Examinaremos essa questão no próximo exercício.

Exercício 19: Uma árvore de decisão no scikit-learn

Neste exercício, usaremos os dados do estudo de caso para criar uma árvore de decisão e especificaremos a profundidade máxima. Também empregaremos uma funcionalidade útil para visualizar a árvore. Para fazê-lo, você terá de executar `conda install graphviz` e `conda install python-graphviz` na linha de comando para instalar a interface Python para o Graphviz. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante dos Exercícios 19 a 21 e da Atividade 5 foram carregados em um Jupyter Notebook que pode ser encontrado em <http://bit.ly/2GI7fJB>. Você pode rolar para a seção apropriada dentro do Jupyter Notebook para localizar o exercício ou a atividade que deseja.

1. Inicie um novo notebook para este capítulo. Para começar, carregue todos os pacotes que usamos, e mais um, `graphviz`, para podermos visualizar árvores de decisão:

```
import numpy as np #cálculo numérico
import pandas as pd #preparação de dados
import matplotlib.pyplot as plt #pacote de plotagem
#A linha a seguir ajuda a renderizar plotagens
%matplotlib inline
import matplotlib as mpl #adiciona o recurso de plotagem
```

```
mpl.rcParams['figure.dpi'] = 400 #figuras em alta resolução
import graphviz #para visualizar árvores de decisão
```

2. Carregue os dados limpos do estudo de caso:

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

Nota: O local dos dados limpos pode ser diferente dependendo de onde você os salvou.

3. Obtenha uma lista com os nomes de colunas do DataFrame:

```
features_response = df.columns.tolist()
```

4. Crie uma lista de colunas para remover as que não sejam características ou a variável de resposta:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school', 'none',
                   'others', 'university']
```

5. Use uma list comprehension para remover esses nomes de colunas de nossa lista com as características e a variável de resposta:

```
features_response = [item for item in features_response if item not in items_to_remove]
features_response
```

Esse código deve exibir a lista com as características e a variável de resposta:

```
['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

Figura 5.2 – Lista com os nomes das características e da variável de resposta.

Agora as características estão preparadas para uso. Em seguida, faremos algumas importações a partir do scikit-learn. Queremos criar uma divisão de treinamento/teste, com a qual já estamos familiarizados. Também queremos importar a classe de árvore de decisão.

6. Execute este código para fazer importações a partir do scikit-learn:

```
from sklearn.model_selection import train_test_split
```

```
from sklearn import tree
```

A biblioteca tree contém as classes relacionadas à árvore de decisão no scikit-learn.

7. Divida os dados em treinamento e teste com o mesmo seed aleatório que usamos em todo o livro para a divisão:

```
X_train, X_test, y_train, y_test = \
train_test_split(df[features_response[:-1]].values, \
df['default payment next month'].values, test_size=0.2, random_state=24)
```

Empregamos todos os elementos da lista, exceto o último, para obter os nomes das características, sem a variável de resposta: `features_response[:-1]`. Fizemos isso para selecionar colunas do DataFrame e recuperar seus valores usando o método `.values`. Fizemos algo semelhante para a variável de resposta, mas especificamos o nome da coluna diretamente. Na criação da divisão de treinamento/teste, usamos o mesmo seed aleatório do trabalho anterior e o mesmo tamanho de divisão. Dessa forma, podemos comparar o trabalho que faremos neste capítulo com resultados anteriores. Especificamente, reservamos o mesmo “conjunto de teste desconhecido” do processo de desenvolvimento de modelo que executamos até aqui.

Estamos prontos para instanciar a classe de árvore de decisão.

8. Instancie a classe de árvore de decisão especificando o parâmetro `max_depth` como 2:

```
dt = tree.DecisionTreeClassifier(max_depth=2)
```

Usamos a classe `DecisionTreeClassifier` porque temos um problema de classificação. Já que especificamos `max_depth=2`, quando criarmos a árvore de decisão com os dados do estudo de caso ela crescerá até uma profundidade de no máximo 2. Agora treinaremos esse modelo.

9. Use este código para ajustar o modelo de árvore de decisão e criar a árvore:

```
dt.fit(X_train, y_train)
```

O código deve exibir a seguinte saída:

```
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=2,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort=False, random_state=None,
splitter='best')
```

Figura 5.3 – Saída do ajuste de um classificador de árvore de decisão.

Na saída de ajuste do modelo é possível ver todas as opções que podem ser definidas com uma árvore de decisão; deixamos a maioria com suas configurações padrão – vamos discuti-las em breve. Por enquanto, ajustamos esse modelo de árvore de decisão para poder usar o pacote `graphviz` e exibir uma representação gráfica da árvore.

10. Exporte o modelo treinado em um formato que seja lido pelo pacote `graphviz` usando este código:

```
dot_data = tree.export_graphviz(dt, out_file=None, filled=True,
                                rounded=True, feature_names=features_response[:-1],
                                proportion=True, class_names=['Not defaulted', 'Defaulted'])
```

Aqui, especificamos várias opções para o método `.export_graphviz`. Primeiro, tivemos de informar que modelo treinado queremos representar, o que foi indicado com o objeto `dt`. Em seguida, informamos que não queremos um arquivo de saída: `out_file=None`. Em vez dele, fornecemos a variável `dot_data` para armazenar a saída desse método. O resto das opções foi usado da seguinte forma:

`filled=True`: Cada nó será preenchido com uma cor.

`rounded=True`: Os nós aparecerão com bordas arredondadas, e não com retângulos.

`feature_names=features_response[:-1]`: Os nomes das características de nossa lista serão usados em vez de nomes genéricos como `X[0]`.

`proportion=True`: A proporção de amostras de cada nó será exibida (discutiremos isso melhor posteriormente).

`class_names=['Not defaulted', 'Defaulted']`: O nome da classe prevista será exibido para cada nó.

Qual é a saída desse método?

Se você examinar o conteúdo de `dot_data`, verá que ele é composto de uma longa string de texto. O pacote `graphviz` interpretará essa string de texto para criar uma visualização.

11. Use o método `.Source` do pacote `graphviz` para criar uma imagem de `dot_data` e exibi-la:

```
graph = graphviz.Source(dot_data)
graph
```

A saída pode ser visualizada na Figura 5.4.

A representação gráfica da árvore de decisão da *Figura 5.4* deve ser renderizada diretamente em seu Jupyter notebook.

Nota: Alternativamente, você poderia salvar a saída de `.export_graphviz` em disco fornecendo um caminho de arquivo para o argumento de palavra-chave `out_file`. Para transformar esse arquivo de saída em um arquivo de imagem, por exemplo, um arquivo `.png` para usar em uma apresentação, você poderia executar este código na linha de comando, substituindo os nomes de arquivo conforme apropriado: `$ dot -Tpng <nome_do_arquivo_exportado> -o <nome_do_arquivo_de_imagem_desejado>.png`

Para ver mais detalhes sobre as opções de `.export_graphviz`, consulte a documentação do scikit-learn (https://scikit-learn.org/stable/modules/generated/sklearn.tree.export_graphviz.html).

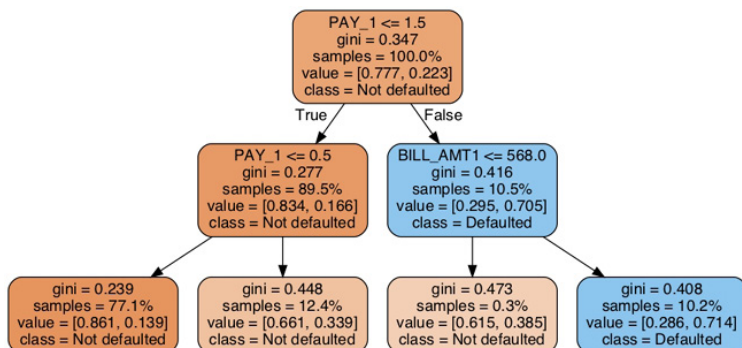


Figura 5.4 – Plotagem da árvore de decisão feita com o graphviz.

A visualização da *Figura 5.4* contém muitas informações sobre como a árvore de decisão foi treinada e como ela pode ser usada para fazer previsões. Discutiremos o processo de treinamento com mais detalhes posteriormente, mas é suficiente dizer que o treinamento de uma árvore de decisão começa com todas as amostras de treinamento no nó inicial do topo da árvore e sua divisão em dois grupos de acordo com um **limite** em uma das características. O ponto de corte é representado por uma condição booleana no nó `PAY_1 <= 1.5` superior.

Todas as amostras em que o valor da característica `PAY_1` é menor ou igual ao ponto de corte de 1.5 serão representadas como `True` na condição booleana. Como mostrado na *Figura 5.4*, essas amostras são classificadas no lado esquerdo da árvore, seguindo a seta que exhibe “`True`” perto dela.

Como você pode ver no gráfico, cada nó que é dividido contém os critérios de divisão na primeira linha de texto. A linha seguinte faz referência a “`gini`”, que discutiremos em breve.

Na linha posterior à linha `gini`, há informações sobre a proporção

de amostras de cada nó. No nó superior, estamos começando com todas as amostras (“samples = 100.0%”). Após a primeira divisão, 89,5% das amostras são classificadas no nó da esquerda, enquanto as 10,5% restantes vão para o nó da direita. Essas informações são exibidas diretamente na visualização e refletem como os dados de treinamento foram usados para criar a árvore. Confirmaremos isso examinando os dados de treinamento.

12. Para confirmar a proporção de amostras de treinamento em que a característica PAY_1 é menor ou igual a 1.5, primeiro identificaremos o índice dessa característica na lista de nomes de características features_response[:-1]:

```
features_response[:-1].index('PAY_1')
```

Esse código deve exibir o seguinte:

4

Figura 5.5 – Índice da característica PAY_1 na lista de nomes de características.

13. Agora obtenha a dimensão dos dados de treinamento:

```
X_train.shape
```

A saída deve ser:

(21331, 17)

Figura 5.6 – Dimensão dos dados de treinamento.

Para confirmar a fração de amostras após a primeira divisão da árvore de decisão, temos de saber a proporção de amostras em que a característica PAY_1 atende a condição booleana usada. Para obter essa informação, podemos usar o índice da característica PAY_1 nos dados de treinamento, correspondente ao índice na lista de nomes de características, e o número de amostras do mesmo conjunto de dados, que é o número de linhas que observamos com .shape.

14. Use este código para confirmar a proporção de amostras após a primeira divisão da árvore de decisão:

```
sum(X_train[:,4] <= 1.5)/X_train.shape[0]
```

A saída deve ser a seguinte:

0.8946134733486475

Figura 5.7 – Proporção de amostras de treinamento em que PAY_1 > =

Aplicando uma condição lógica à coluna dos dados de treinamento correspondente à característica `PAY_1`, e obtendo a soma disso, calculamos o número de amostras que a atendem. Em seguida, dividindo pelo número total de amostras, convertemos o resultado em uma proporção. Podemos ver que a proporção que calculamos diretamente a partir dos dados de treinamento é igual à proporção exibida no nó esquerdo após a primeira divisão na *Figura 5.4*.

Após a primeira divisão, as amostras contidas em cada um dos dois nós do primeiro nível são divididas novamente. À medida que novas divisões forem feitas depois da primeira divisão, proporções cada vez menores dos dados de treinamento serão atribuídas a nós específicos nos níveis subsequentes de um galho, como pode ser visto na *Figura 5.4*.

Agora queremos interpretar as linhas de texto restantes dos nós da *Figura 5.4*. As linhas que começam com “value” fornecem as frações de classes da variável de resposta para as amostras contidas em cada nó. Por exemplo, no nó superior, vemos “value = [0.777, 0.223]”. Isso é simplesmente a fração de classes do conjunto de treinamento total, que podemos confirmar na próxima etapa.

15. Calcule a fração de classes do conjunto de treinamento com este código:

```
np.mean(y_train)
```

A saída deve ser:

0.223102526838873

Figura 5.8 – Fração de classes de amostras positivas nos dados de treinamento.

Esse valor é igual ao segundo membro do par de números que vêm após “value” no nó superior; o primeiro número é apenas um menos esse valor, em outras palavras, a fração de amostras de treinamento negativas. Em cada nó subsequente, a fração de classes das amostras que estiverem contidas apenas nesse nó será exibida. As frações de classes também mostram como os nós serão coloridos: os que tiverem maior proporção da classe negativa serão laranja, com laranja escuro significando proporções mais altas, enquanto os de maior proporção da classe positiva terão um esquema semelhante usando a cor azul.

Para concluir, a linha que começa com “class” indica como a árvore

de decisão fará previsões a partir de um nó específico, se ele for um nó folha. Árvores de decisão usadas para classificação fazem previsões determinando em que nó folha uma amostra será classificada, dados os valores das características, e prevendo a classe da maioria das amostras de treinamento nesse nó folha. Essa estratégia significa que as proporções das classes em cada nó são as informações necessárias para fazermos uma previsão.

Por exemplo, se não fizermos divisões e formos forçados a fazer uma previsão não sabendo nada exceto as frações de classes dos dados de treinamento totais, escolheremos simplesmente a classe majoritária. Já que a maioria das pessoas não está inadimplente, a classe do nó superior será “Not defaulted”. No entanto, as frações de classes dos nós de níveis mais profundos serão diferentes, levando a outras previsões. Discutiremos o processo de treinamento na próxima seção.

Importância de `max_depth`

Você deve lembrar-se de que o único hiperparâmetro que especificamos neste exercício foi `max_depth`, isto é, a profundidade máxima até a qual a árvore de decisão pode crescer durante o processo de treinamento do modelo. Por acaso esse é um dos hiperparâmetros mais importantes. Se não impusermos um limite para a profundidade, a árvore crescerá até que uma das outras limitações, especificadas por outros hiperparâmetros, surta efeito. Isso pode levar a árvores muito profundas, com vários nós. Por exemplo, considere quantos nós folhas poderia haver em uma árvore com profundidade 20. Seriam 2^{20} nós folhas, ou seja, mais de 1 milhão! Temos 1 milhão de amostras de treinamento para classificar em todos esses nós? Nesse caso, não. Seria claramente impossível criar essa árvore usando esses dados de treinamento e com cada nó antes do nível final sendo dividido. No entanto, se removermos o limite `max_depth` e reexecutarmos o treinamento do modelo deste exercício, observe o efeito:



Figura 5.9 – Parte da árvore de decisão criada sem profundidade máxima.

Aqui, exibimos uma parte da árvore de decisão que foi criada com as opções padrão, que incluem `max_depth=None`, significando que não

há limitação para a profundidade da árvore. A árvore inteira é cerca de duas vezes mais larga que a porção mostrada aqui. Há tantos nós que eles só aparecem como manchas (patches) laranja ou azuis muito pequenas; a interpretação exata de cada nó não é importante, já que só estamos tentando ilustrar como as árvores podem ficar grandes. Deve ter ficado claro que, sem hiperparâmetros para controlar o processo de crescimento, podemos ter árvores extremamente grandes e complexas.

Treinando árvores de decisão: Impureza dos nós

Até agora, tratamos o processo de treinamento da árvore de decisão como uma caixa preta. Você sabe apenas que uma árvore de decisão faz previsões usando características e as frações de classes das amostras de treinamento dos nós folhas.

Como definir as divisões durante o processo de treinamento?

Dado que o método de previsão usa a classe majoritária de um nó folha, faz sentido querermos encontrar nós folhas que sejam principalmente de uma classe ou outra. Em uma situação ideal, os dados de treinamento seriam divididos de modo que cada nó folha contivesse amostras totalmente positivas ou totalmente negativas. Assim teríamos um alto nível de confiança de que, após classificada em um desses nós, uma nova amostra seria positiva ou negativa. Na prática, isso raramente acontece, se é que ocorre. No entanto, ilustra o objetivo do treinamento de árvores de decisão – ou seja, criar divisões de modo que os próximos dois nós tenham uma **pureza** mais alta, ou, em outras palavras, fiquem mais próximos de conter amostras somente positivas ou negativas.

Na verdade, as árvores de decisão são treinadas com o uso do inverso da pureza, ou a **impureza dos nós**. Trata-se de uma medida de quanto o nó está longe de ter 100% das amostras de treinamento pertencendo a uma única classe e é análoga ao conceito de função custo, que indica quanto uma solução específica está longe de ser uma solução teórica perfeita. O conceito mais intuitivo da impureza dos nós é a taxa de classificação errônea. Adotando uma notação amplamente usada (por exemplo, a encontrada em <https://scikit-learn.org/stable/modules/tree.html>) para a proporção de amostras em cada nó pertencentes a uma classe específica, podemos definir p_{mk} como a

proporção de amostras pertencentes à k ésima classe do m ésimo nó. Em um problema de classificação binária há apenas duas classes: $k = 0$ e $k = 1$. Para um nó m específico, a **taxa de classificação errônea** é simplesmente a proporção da classe menos comum nesse nó, já que todas essas amostras serão classificadas erroneamente quando a classe majoritária do nó for usada na previsão.

Visualizaremos a taxa de classificação errônea como uma maneira de começar a pensar em como as árvores de decisão são treinadas. Consideraremos programaticamente as possíveis frações, p_{m0} , entre 0.01 e 0.99 da classe negativa, $k = 0$, em um nó, m , usando a função `linspace` do NumPy:

```
pm0 = np.linspace(0.01,0.99,99)
```

A fração da classe positiva desse nó será então composta simplesmente do resto das amostras:

$$p_{m1} = 1 - p_{m0}$$

Figura 5.10 – Equação para o cálculo da fração de classe positiva para o nó $m0$.

A taxa de classificação errônea desse nó corresponderá a qualquer que seja a menor fração de classe, entre **pm0** e **pm1**. Podemos encontrar o menor dos elementos correspondentes entre dois arrays com a mesma dimensão no NumPy usando a função `minimum`:

```
misclassification_rate = np.minimum(pm0, pm1)
```

Como ficaria a taxa de classificação errônea quando plotada em relação às possíveis frações da classe negativa?

Podemos plotar isso usando o código a seguir:

```
mpl.rcParams['figure.dpi'] = 400
plt.plot(pm0, misclassification_rate, label='Misclassification rate')
plt.xlabel('$p_{m0}$')
plt.legend()
```

Você deve obter este gráfico:

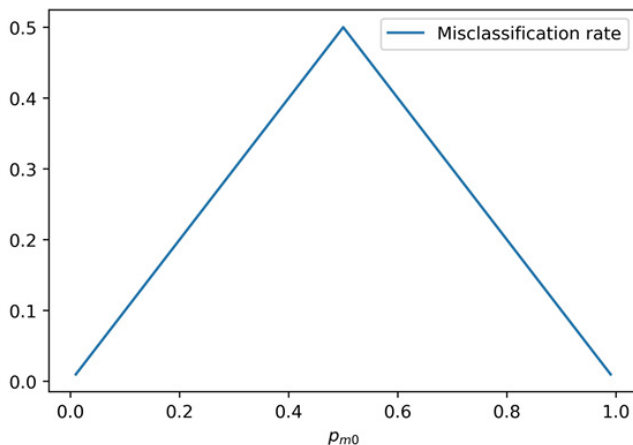


Figura 5.11 – Taxa de classificação errônea de um nó.

Ficou claro que, quanto mais próxima a fração da classe negativa, p_{m0} , estiver de 0 ou 1, menor será a taxa de classificação errônea. Como essa informação é usada na criação de árvores de decisão? Acompanhemos o processo.

Sempre que um nó é dividido na criação de uma árvore de decisão, dois novos nós são criados. Já que a previsão feita a partir de um desses nós considerará apenas a classe majoritária, um objetivo importante será reduzir a taxa de classificação errônea. Logo, queremos encontrar uma característica, entre todas as possíveis, e selecionar um valor dela onde criar um ponto de corte de modo que a taxa de classificação errônea nos dois novos nós seja a mais baixa possível na média de todas as classes. Isso chega muito próximo do processo real que é usado para treinar árvores de decisão.

Continuando por enquanto com a ideia de reduzir a taxa de classificação errônea, o algoritmo de treinamento de árvores de decisão executa a divisão de nós considerando todas as características. No entanto, ele pode considerar apenas um subconjunto selecionado aleatoriamente se você configurar o hiperparâmetro `max_features` com algo menor que o número total de características. Discutiremos as possíveis razões para se fazer isso posteriormente. Nos dois casos, o algoritmo considerará cada limite possível para cada característica candidata e selecionará a que resultar na menor impureza, calculada como a impureza média nos dois novos nós ponderada pelo número de amostras de cada nó. O processo de divisão de nós é mostrado na *Figura 5.12*. Ele é repetido até um critério de parada definido para a árvore, como `max_depth`, ser atendido:



Figura 5.12 – Como selecionar uma característica e um limite para dividir um nó.

Embora a taxa de classificação errônea seja uma medida de impureza intuitiva, há medidas melhores que podem ser usadas para encontrarmos divisões durante o processo de treinamento do modelo. As duas opções disponíveis no scikit-learn para cálculo da impureza, que você pode especificar com o argumento de palavra-chave `criterion`, são a **impureza de Gini** e a **entropia cruzada**. Vamos descrevê-las matematicamente e compará-las em relação à taxa de classificação errônea.

A impureza de Gini é calculada para um nó m usando a fórmula a seguir:

$$\text{Gini} = \sum_k p_{mk} (1 - p_{mk})$$

Figura 5.13 – Equação para o cálculo da impureza de Gini.

Aqui, a soma é obtida com todas as classes. No caso de um problema de classificação binária, há somente duas classes e podemos escrever isso programaticamente da seguinte forma:

$$\text{gini} = (\text{pm0} * (1 - \text{pm0})) + (\text{pm1} * (1 - \text{pm1}))$$

A entropia cruzada é calculada com o uso desta fórmula:

$$\text{entropia cruzada} = - \sum_k p_{mk} \log(p_{mk})$$

Figura 5.14 – Equação para o cálculo da entropia cruzada.

Usando este código, podemos calcular a entropia cruzada:

$$\text{cross_ent} = -1 * ((\text{pm0} * \text{np.log}(\text{pm0})) + (\text{pm1} * \text{np.log}(\text{pm1})))$$

Para adicionar a impureza de Gini e a entropia cruzada à plotagem da taxa de classificação errônea e compará-las, temos apenas de incluir as linhas de código a seguir após plotar a taxa:

```
plt.plot(pm0, gini, label='Gini impurity')
plt.plot(pm0, cross_ent, label='Cross entropy')
```


A plotagem final deve ficar assim:

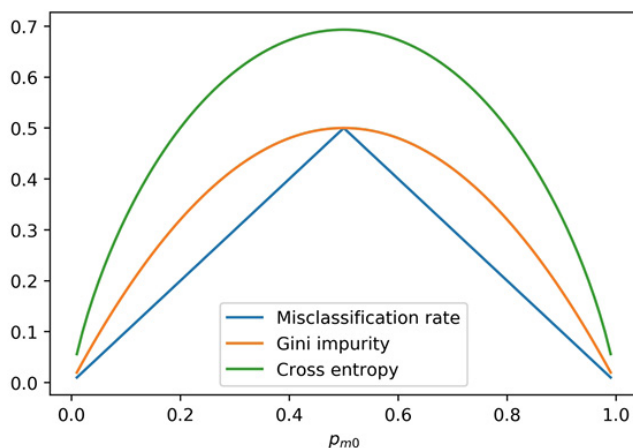


Figura 5.15 – Taxa de classificação errônea, impureza de Gini e entropia cruzada.

Como a taxa de classificação errônea, tanto a impureza de Gini quanto a entropia cruzada são mais altas quando as frações de classes são iguais a 0.5 e diminuem à medida que o nó se torna mais puro – em outras palavras, quando eles contêm uma proporção mais alta de apenas uma das classes. No entanto, a impureza de Gini é um pouco mais íngreme que a taxa de classificação errônea em certas regiões da fração de classes, o que lhe permite encontrar mais eficientemente a melhor divisão. A entropia cruzada parece ainda mais íngreme. Então, qual é a melhor para o nosso trabalho? Esse é o tipo de pergunta que não tem uma resposta concreta para todos os datasets. Você deve considerar as duas métricas de impureza em uma busca de hiperparâmetros por validação cruzada para determinar a apropriada. É bom ressaltar que, no scikit-learn, a impureza de Gini pode ser especificada com o argumento `criterion` usando a string 'gini', enquanto a entropia cruzada é representada apenas por 'entropy'.

Características usadas para as primeiras divisões: Conexões com a seleção de características univariada e as interações

Podemos começar a ter uma ideia de como várias características são importantes para os modelos de árvore de decisão de acordo com a pequena árvore mostrada na *Figura 5.4*. Observe que `PAY_1` foi a característica selecionada para a primeira divisão. Isso significa que ela é a melhor característica em termos de diminuição da impureza do

nó que contém todas as amostras de treinamento. Se nos lembrarmos de nossa experiência com a seleção de características univariada mostrada no *Capítulo 3, Detalhes da regressão logística e exploração de características*, PAY_1 foi a principal característica selecionada no teste F. Logo, o uso dessa característica na primeira divisão da árvore de decisão faz sentido segundo nossa análise anterior.

No segundo nível da árvore, há outra divisão em PAY_1, assim como uma divisão em BILL_AMT_1. BILL_AMT_1 não foi listada entre as principais características na seleção univariada. No entanto, pode haver uma interação importante entre BILL_AMT_1 e PAY_1 que não seja detectada por métodos univariados. Especificamente, nas divisões selecionadas pela árvore de decisão, parece que contas com valor maior ou igual a 2 para PAY_1 e com BILL_AMT_1 maior que 568 correm mais risco de inadimplência. Esse efeito combinado de PAY_1 e BILL_AMT_1 é uma interação e também pode ser o motivo pelo qual conseguimos melhorar o desempenho da regressão logística ao incluir termos de interação na atividade do capítulo anterior.

Treinando árvores de decisão: Um algoritmo ganancioso

Não há garantias de que uma árvore de decisão treinada pelo processo descrito anteriormente será a melhor árvore possível para encontrarmos os nós folhas de menor impureza. Isso ocorre porque o algoritmo usado para treinar árvores de decisão é o que chamamos de algoritmo ganancioso. Nesse contexto, significa que, a cada oportunidade de divisão de um nó, o algoritmo procura a melhor divisão possível naquele momento, sem considerar o fato de que as oportunidades de divisões posteriores estão sendo afetadas.

Por exemplo, considere o cenário hipotético a seguir: a melhor divisão inicial para os dados de treinamento do estudo de caso envolve PAY_1, como vimos na *Figura 5.4*. Porém, e se dividíssemos em BILL_AMT_1 e fizéssemos divisões subsequentes em PAY_1 no próximo nível? Isso significaria que a impureza dos nós folhas seria menor? Ainda que a divisão inicial em BILL_AMT_1 não seja a princípio a melhor divisão disponível, o resultado final pode ser melhor se agirmos dessa forma. O algoritmo não tem como encontrar soluções como essa, já que só considera a melhor divisão possível em cada nó. Quero deixar claro que esse é apenas um cenário hipotético e não temos razão para suspeitar que ocorrerá com os dados do estudo de caso.

A razão para ainda usarmos o algoritmo que descrevemos

anteriormente é porque ele demora significativamente mais para considerar todas as divisões possíveis de modo que permite encontrar a árvore ótima. Apesar dessa deficiência do processo de treinamento da árvore de decisão, há métodos que você pode usar para reduzir os possíveis efeitos danosos do algoritmo ganancioso. Em vez de procurarmos a melhor divisão em cada nó, o argumento de palavra-chave `splitter` da classe de árvore de decisão pode ser especificado como `random` para selecionarmos uma característica aleatória na qual fazer a divisão. No entanto, o padrão é `best`, que procura a melhor divisão em todas as características. Outra opção, que já discutimos, é limitar o número de características usando a palavra-chave `max_features`, que será consultada a cada oportunidade de divisão. Para concluir, você também pode usar combinações de árvores de decisão como as florestas aleatórias, que descreveremos em breve. É bom ressaltar que, além de evitar os efeitos prejudiciais do algoritmo ganancioso, essas opções também resolvem o `overfitting` pelo qual as árvores de decisão são frequentemente criticadas.

Já examinamos o uso do parâmetro `max_depth` como limite de até que profundidade uma árvore crescerá. Contudo, há várias outras opções disponíveis no `scikit-learn`. Elas estão relacionadas principalmente com quantas amostras estarão presentes em um nó folha ou quanto a impureza pode ser diminuída pela continuação da divisão dos nós. Como já mencionamos, o tamanho de seu dataset pode restringi-lo em termos de até que profundidade você poderá estender uma árvore. E talvez não faça sentido criar árvores mais profundas, especialmente se o processo de divisão não estiver mais encontrando nós com pureza mais alta.

Notas e razões para a seleção		
<code>fit_max_depth</code>	Parâmetro usado para controlar a impureza do nó	
<code>fit_min_samples_leaf</code>	Indica o número mínimo de amostras em todas as características dos candidatos ao fazer uma divisão, ou se selecionaremos uma aleatoriamente.	
<code>fit_max_depth</code>	Parâmetro usado para controlar a profundidade máxima de crescimento da árvore, embora ela possa ser interrompida por razões especificadas em outros hiperparâmetros. Um inteiro int significa interromper o crescimento da árvore após esse número de níveis.	
<code>fit_min_samples_split</code>	Se for um inteiro, um nó deve ter pelo menos esse número de amostras para ser dividido. Se for um float, ele será a fração do número total de amostras que devem existir em um nó para que ele seja dividido.	
<code>fit_min_samples_leaf</code>	Semelhante a <code>min_samples_split</code> , mas refere-se ao número de	

Não há interações e relacionamentos lineares

Já que cada divisão sucessiva de uma árvore de decisão é executada em um subconjunto das amostras resultantes de divisões anteriores, as árvores de decisão podem descrever relacionamentos não lineares complexos de uma única característica, assim como as interações entre elas. Considere nossa discussão anterior de por que BILL_AMT_1 foi selecionada no segundo nível da árvore na *Figura 5.4*. Além disso, como um exemplo hipotético com dados sintéticos, considere o dataset a seguir para a classificação:



Figura 5.17 – Exemplo de dataset de classificação, com as classes sendo exibidas em vermelho e azul.

Sabemos pelo *Capítulo 3, Detalhes da regressão logística e exploração de características* que a regressão logística tem um limite de decisão linear. Como você acha que a regressão logística pode lidar com um dataset como o mostrado na *Figura 5.17*? Onde você deve desenhar uma linha para separar as classes azul e vermelha? Deve ter ficado claro que, sem a criação de características adicionais por engenharia, provavelmente a regressão logística não será um bom classificador para esses dados. Agora pense no conjunto de regras “se, então” de uma árvore de decisão, que poderia ser usada com as características representadas nos eixos x e y da *Figura 5.17*. Você acha que uma árvore de decisão seria eficaz com esses dados?

Aqui, plotamos no plano de fundo as probabilidades previstas de associação de classe usando as cores vermelha e azul para estes dois modelos:

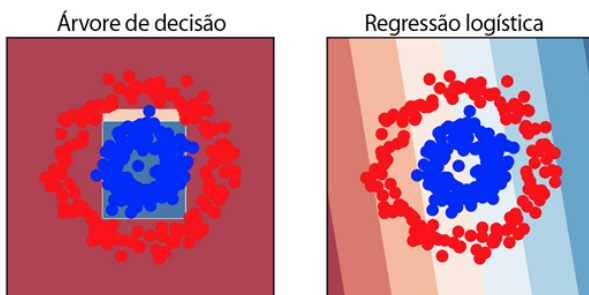


Figura 5.18 – Previsões da árvore de decisão e da regressão logística.

Na *Figura 5.18*, as probabilidades previstas dos dois modelos foram coloridas de modo que o vermelho mais escuro corresponda a uma probabilidade prevista mais alta para a classe vermelha, enquanto o azul mais escuro corresponde a uma probabilidade mais alta para a classe azul. Podemos ver que a árvore de decisão isola a classe azul no meio do círculo de pontos vermelhos. Isso ocorre porque, usando limites para as coordenadas x e y no processo de divisão de nós, uma árvore de decisão pode modelar matematicamente o fato de que a localização das classes azul e vermelha depende das coordenadas x e y juntas (interações) e que a probabilidade de cada classe não é uma função linearmente crescente ou decrescente de x ou y (não linearidades). Consequentemente, a abordagem de árvore de decisão deve executar a maioria das classificações de maneira correta.

No entanto, a regressão logística tem um limite de decisão linear, que é a linha reta entre as manchas azuis e vermelhas mais claras no plano de fundo. O limite de decisão da regressão logística passa exatamente pelo meio dos dados e não fornece um classificador útil. Isso mostra o poder das árvores de decisão “prontas para uso”, sem a necessidade de engenharia de características não lineares ou de interação.

Probabilidades previstas

Você deve estar se perguntando como as probabilidades previstas exibidas na *Figura 5.18* foram obtidas pelo modelo de árvore de decisão. Sabemos que a regressão logística produz probabilidades como saída bruta. No entanto, uma árvore de decisão faz previsões com base na classe majoritária dos nós folhas. De onde vem essa probabilidade? Na verdade, as árvores de decisão oferecem o método `.predict_proba` do `scikit-learn`. A probabilidade é baseada na proporção da classe majoritária no nó folha. Se o nó folha for 75% composto da classe positiva, por exemplo, a previsão para esse nó será constituída pela classe positiva e a probabilidade prevista será de 0,75. As

probabilidades previstas a partir de árvores de decisão não são consideradas tão estatisticamente rigorosas quanto as de modelos lineares generalizados, mas mesmo assim costumam ser usadas na avaliação do desempenho de modelos por métodos que dependem da variação do limite para a classificação, como a curva ROC ou a curva precision-recall.

Nota: Estamos nos concentrando nas árvores de decisão aqui para a classificação devido à natureza do estudo de caso. Porém, elas também podem ser usadas para a regressão, o que as torna um método versátil. O processo de criação da árvore para a regressão é semelhante ao da classificação, exceto por, em vez de tentar reduzir a impureza dos nós, a árvore da regressão reduzir outras métricas como o erro quadrático médio (MSE, mean squared error) ou o erro absoluto médio (MAE, mean absolute error) das previsões, em que a previsão para um nó pode ser a média ou a mediana das amostras do nó, respectivamente.

Abordagem mais conveniente da validação cruzada

No *Capítulo 4, O trade-off entre viés e variância*, conhecemos detalhadamente a validação cruzada escrevendo nossa própria função para executá-la e usando a classe `KFold` para gerar os índices de treinamento e teste. No entanto, o `scikit-learn` oferece uma classe conveniente que se encarrega de grande parte do trabalho pesado para nós: `GridSearchCV`. `GridSearchCV` pode receber como entrada o modelo para o qual queremos encontrar hiperparâmetros ótimos, como uma árvore de decisão ou uma regressão logística, e a “grade” de hiperparâmetros na qual queremos executar a validação cruzada. Por exemplo, em uma regressão logística, poderíamos querer obter o resultado da média da validação cruzada de todos os folds para diferentes valores do parâmetro de regularização `C`. Com as árvores de decisão, poderíamos querer explorar diferentes profundidades de árvores. Você também pode pesquisar vários hiperparâmetros ao mesmo tempo, por exemplo, se quiser testar diferentes profundidades de árvores e diferentes números para `max_features` em cada divisão de nó.

`GridSearchCV` executa o que é chamado de busca exaustiva na grade com todas as combinações de parâmetros possíveis fornecidos. Isso significa que, se fornecêssemos cinco valores diferentes para dois hiperparâmetros, o procedimento de validação cruzada seria

executado $5 \times 5 = 25$ vezes. Se você pesquisar vários valores envolvendo muitos hiperparâmetros, o número de execuções da validação cruzada pode crescer rapidamente. Nesses casos, pode ser melhor usar RandomizedSearchCV, que pesquisa um número de combinações de hiperparâmetros selecionado aleatoriamente a partir do universo de possibilidades fornecidas para a grade.

GridSearchCV pode acelerar nosso trabalho otimizando o processo de validação cruzada. Você deve estar familiarizado com os conceitos de validação cruzada porque os vimos no capítulo anterior, logo, prosseguiremos diretamente para a listagem de todas as opções que estão disponíveis para GridSearchCV. No exercício a seguir, passaremos à prática usando GridSearchCV com os dados do estudo de caso para pesquisar hiperparâmetros para um classificador de árvore de decisão. Aqui estão as opções de GridSearchCV:

Notas e razões para uma seleção		
<code>GridSearchCV</code>	GridSearchCV é instanciado a partir de uma classe de modelo. Os hiperparâmetros serão atualizados quando GridSearchCV executar sua tarefa.	
<code>fit_params</code>	GridSearchCV aceita todos os nomes dos parâmetros como chaves e as listas de parâmetros como valores. Estes últimos serão os valores dos hiperparâmetros para os quais você quiser buscar todas as combinações possíveis. Para fazer múltiplas buscas na grade, forneça uma lista de dicionários.	
<code>scoring</code>	Representa a métrica de avaliação de modelos que você deseja usar para avaliar o desempenho de treinamento e teste entre os folds, por exemplo, 'roc_auc'.	
<code>n_jobs</code>	Indica o número de jobs de processamento que serão executados em paralelo. Pode acelerar a validação cruzada para a execução de jobs paralelos, mas é bom testar para garantir.	
<code>use_cross_validation</code>	Indica o número de jobs ou a fórmula do número de jobs a serem despachados. É relevante para o processamento paralelo com o uso de n_jobs.	
<code>iid</code>	Indica se os dados são independentes e identicamente distribuídos (i.i.d.). Use True para calcular o resultado da média ponderada entre os folds, empregando o número de amostras de cada fold para definir o peso. Use False para calcular uma média não ponderada. Não é relevante se o número de amostras for o mesmo em cada fold.	
<code>cv</code>	Se o sistema de validação cruzada for genérico, este será o número de folds usados na validação cruzada.	
<code>refit</code>	Após a execução da validação cruzada, os "melhores" hiperparâmetros de acordo com a métrica especificada em scoring poderão ser usados diretamente com o objeto GridSearchCV ajustado para a criação de previsões. Se refit=True, o modelo será reajustado para todos os dados (e não apenas para os de um dos folds) com o uso dos melhores hiperparâmetros. Utilize o argumento de string se várias métricas forem especificadas.	
<code>verbose</code>	Indica o volume da saída que você verá no procedimento de validação cruzada.	
<code>best_score_only</code>	Define se ou não deve ser feito se um erro ocorrer durante o ajuste do modelo.	
<code>return_train_score</code>	Determina se calcularemos e retornaremos os resultados do treinamento nos folds. Não é necessário na seleção dos melhores hiperparâmetros com base nos resultados de testes com os folds, e para alguns datasets e modelos pode causar uma demora maior. No entanto, fornece insights sobre possível overfitting.	

Figura 5.19 – Opções de GridSearchCV.

Exercício 20: Encontrando hiperparâmetros ótimos para uma árvore de decisão

Neste exercício, usaremos GridSearchCV para ajustar os hiperparâmetros de um modelo de árvore de decisão. Você conhecerá uma maneira conveniente de pesquisar diferentes hiperparâmetros com o scikit-learn. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook e podem ser encontrados em <http://bit.ly/2GI7fJB>.

1. Importe a classe GridSearchCV com este código:

```
from sklearn.model_selection import GridSearchCV
```

A próxima etapa é definir os hiperparâmetros que queremos pesquisar usando a validação cruzada. Encontraremos a melhor profundidade máxima da árvore usando o parâmetro `max_depth`. Árvores mais profundas têm mais divisões de nós, que particionam o conjunto de treinamento em subespaços cada vez menores usando as características. Não sabemos a melhor profundidade máxima antecipadamente, mas é útil considerar alguns casos limite ao pensar no intervalo de parâmetros usado na busca na grade.

Sabemos que um é a profundidade mínima, consistindo em uma árvore com apenas uma divisão. No que diz respeito à maior profundidade, você pode considerar quantas amostras há em seus dados de treinamento, ou mais apropriadamente nesse caso, quantas amostras existirão no fold de treinamento para cada divisão da validação cruzada. Executaremos uma validação cruzada com 4 folds como fizemos no capítulo anterior. Logo, quantas amostras teremos no fold de treinamento e que relação isso tem com a profundidade da árvore?

2. Encontre o número de amostras dos dados de treinamento **usando este código**:

```
X_train.shape
```

A saída deve ser:

```
(21331, 17)
```

Figura 5.20 – Dimensão dos dados de treinamento.

Com 21.331 amostras de treinamento e uma validação cruzada com 4 folds, haverá três quartos das amostras, ou cerca de 16.000 amostras, em cada fold de treinamento.

O que isso significa para a profundidade que daremos à árvore?

Uma limitação teórica é a de que precisamos de pelo menos uma amostra em cada folha. Pela nossa discussão sobre como a profundidade da árvore está relacionada com o número de folhas, sabemos que uma árvore que se divide em cada nó antes do último nível tem 2^n nós folhas. Logo, uma árvore com L nós folhas tem profundidade de aproximadamente $\log_2(L)$. No caso limite, se estendermos a árvore até uma profundidade suficiente para que cada nó folha tenha uma única amostra de treinamento para um fold específico, a profundidade será igual a $\log_2(16,000) \approx 14$. Portanto, 14 é o limite teórico da profundidade da árvore que poderíamos criar nesse caso.

Na prática, provavelmente você não vai querer criar uma árvore com essa profundidade, já que as regras usadas para gerar a árvore de decisão serão muito específicas para os dados de treinamento e o modelo deve sofrer sobreajuste. No entanto, isso nos dá uma ideia do intervalo de valores que podemos considerar para o hiperparâmetro `max_depth`. Examinaremos um intervalo de profundidades de 1 a 12.

3. Defina um dicionário com a chave sendo o nome do hiperparâmetro e o valor sendo a lista de valores desse hiperparâmetro que queremos pesquisar na validação cruzada:

```
params = {'max_depth': [1, 2, 4, 6, 8, 10, 12]}
```

Nesse caso, estamos pesquisando apenas um hiperparâmetro. No entanto, você poderia definir um dicionário com vários pares chave-valor para procurar diversos hiperparâmetros simultaneamente.

Agora queremos instanciar a classe `GridSearchCV`.

4. Instancie a classe `GridSearchCV` usando estas opções:

```
cv = GridSearchCV(dt, param_grid=params, scoring='roc_auc', fit_params=None,
                  n_jobs=None, iid=False, refit=True, cv=4, verbose=1,
                  pre_dispatch=None, error_score=np.nan, return_train_score=True)
```

Observe que pudemos reutilizar o objeto de árvore de decisão, `dt`, que instanciamos anteriormente neste capítulo. Ao criar `dt`, usamos os argumentos padrão para todas as opções exceto `max_depth`. Porém esse hiperparâmetro será redefinido aqui com o uso do dicionário `params` em cada iteração do loop de validação cruzada. As outras opções dignas de nota são o uso da métrica ROC AUC (`scoring='roc_auc'`), a execução de uma validação com 4 folds (`cv=4`) e o cálculo dos resultados do treinamento

(return_train_score=True) para a avaliação do trade-off entre viés e variância.

Uma vez que o objeto de validação cruzada estiver definido, poderemos usar o método .fit nele como faríamos com um objeto de modelo. Isso encapsulará toda a funcionalidade do loop de validação cruzada que criamos no capítulo anterior.

5. Execute uma validação cruzada com 4 folds para procurar a profundidade máxima ótima usando este código:

```
cv.fit(X_train, y_train)
```

A saída deve ser:

```
Fitting 4 folds for each of 7 candidates, totalling 28 fits

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 28 out of 28 | elapsed: 2.8s finished

GridSearchCV(cv=4, error_score=nan,
             estimator=DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=None,
             max_features=None, max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, presort=False, random_state=None,
             splitter='best'),
             fit_params=None, iid=False, n_jobs=None,
             param_grid={'max_depth': [1, 2, 4, 6, 8, 10, 12]},
             pre_dispatch=None, refit=True, return_train_score=True,
             scoring='roc_auc', verbose=1)
```

Figura 5.21 – Saída do ajuste da validação cruzada.

Todas as opções que especificamos foram exibidas como parte da saída. Além disso, há algumas informações na saída sobre quantos ajustes de validação cruzada foram executados. Tínhamos 4 folds e 7 hiperparâmetros, o que totaliza $4 \times 7 = 28$ ajustes. O tempo que gastamos também foi exibido. Você pode controlar o volume de saída que deseja obter desse procedimento com o argumento de palavra-chave verbose; números maiores significam mais saída.

É hora de examinar os resultados do procedimento de validação cruzada. O método cv_results_ é um dos que estão disponíveis no objeto GridSearchCV ajustado. Trata-se de um dicionário contendo os nomes dos resultados como chaves e os resultados propriamente ditos como valores. Por exemplo, a chave mean_test_score contém o resultado da média do teste em todos os folds para cada um dos sete hiperparâmetros. Você poderia examinar essa saída diretamente executando cv.cv_results_ em uma célula de código. No entanto, ela não seria fácil de ler. Dicionários com esse tipo de estrutura podem ser usados imediatamente na criação de um DataFrame do pandas, o que facilita a leitura dos resultados.

6. Execute o código a seguir para criar e examinar um DataFrame do

pandas com resultados de validação cruzada:

```
cv_results_df = pd.DataFrame(cv.cv_results_)  
cv_results_df
```

A saída será a seguinte:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	params	split0_test_score
0	0.023161	0.002917	0.002712	0.000827	1	{'max_depth': 1}	0.639514
1	0.040478	0.005298	0.003616	0.001432	2	{'max_depth': 2}	0.695134
2	0.062975	0.001703	0.002196	0.000017	4	{'max_depth': 4}	0.732720
3	0.094926	0.005329	0.002393	0.000150	6	{'max_depth': 6}	0.743836
4	0.123850	0.008124	0.003107	0.000768	8	{'max_depth': 8}	0.727948
5	0.142454	0.001005	0.002620	0.000221	10	{'max_depth': 10}	0.709049
6	0.178841	0.016180	0.002716	0.000238	12	{'max_depth': 12}	0.675597

Figura 5.22 – Resultados da validação cruzada.

O DataFrame tem uma linha para cada combinação de hiperparâmetros da grade. Já que só estamos pesquisando um hiperparâmetro aqui, há uma linha para cada um dos sete valores que pesquisamos. Podemos ver uma saída extensa para cada linha, como a média e o desvio padrão do tempo em segundos que cada um dos quatro folds demorou tanto no treinamento (ajuste) quanto no teste (resultado). Os valores de hiperparâmetros que foram pesquisados também são exibidos. Na Figura 5.22, podemos ver o resultado da ROC AUC para os dados de teste do primeiro fold (índice 0). Mas quais são as outras colunas do DataFrame de resultados?

7. Visualize os nomes das outras colunas do DataFrame de resultados usando este código:

```
cv_results_df.columns
```

A saída será:

```
Index(['mean_fit_time', 'std_fit_time', 'mean_score_time', 'std_score_time',  
      'param_max_depth', 'params', 'split0_test_score', 'split1_test_score',  
      'split2_test_score', 'split3_test_score', 'mean_test_score',  
      'std_test_score', 'rank_test_score', 'split0_train_score',  
      'split1_train_score', 'split2_train_score', 'split3_train_score',  
      'mean_train_score', 'std_train_score'],  
      dtype='object')
```

Figura 5.23 – Colunas do DataFrame de resultados da validação cruzada.

As colunas do DataFrame de resultados da validação cruzada incluem os resultados do teste para cada fold, sua média e desvio

padrão, e as mesmas informações para os resultados do treinamento.

De modo geral, a “melhor” combinação de hiperparâmetros é a que tem o resultado de média de teste mais alto. Essa é uma estimativa de como o modelo, ajustado com esses hiperparâmetros, se sairia com novos dados. Criaremos uma plotagem exibindo como o resultado de média do teste varia com o hiperparâmetro `max_depth`. Também exibiremos os resultados de média do treinamento na mesma plotagem, para ver como o viés e a variância mudam à medida que permitimos que árvores mais profundas e complexas sejam criadas durante o ajuste do modelo.

Incluiremos os desvios padrão dos resultados de treinamento e teste de 4 folds como barras de erro, usando a função `errorbar` do Matplotlib. Isso dará uma indicação de quanto os resultados variam entre os folds.

8. Execute o código a seguir para criar a plotagem de uma barra de erro dos resultados de treinamento e teste para cada valor de `max_depth` que foi examinado na validação cruzada:

```
ax = plt.axes()
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_train_score'],
            yerr=cv_results_df['std_train_score'],
            label='Mean  $\pm$  1 SD training scores')
ax.errorbar(cv_results_df['param_max_depth'],
            cv_results_df['mean_test_score'],
            yerr=cv_results_df['std_test_score'],
            label='Mean  $\pm$  1 SD testing scores')
ax.legend()
plt.xlabel('max_depth')
plt.ylabel('ROC AUC')
```

A plotagem pode ser visualizada na Figura 5.23.

Os desvios padrão dos resultados de treinamento e teste são exibidos como linhas verticais em cada valor de `max_depth` que foi testado; a distância acima e abaixo do resultado de média tem desvio padrão 1. Na criação de plotagens de barra de erro, é melhor assegurar que as unidades de medida de erro sejam iguais às unidades do eixo y. Nesse caso, elas são, já que o desvio padrão tem as mesmas unidades dos dados subjacentes, ao contrário da variância, por exemplo, que tem unidades quadradas.

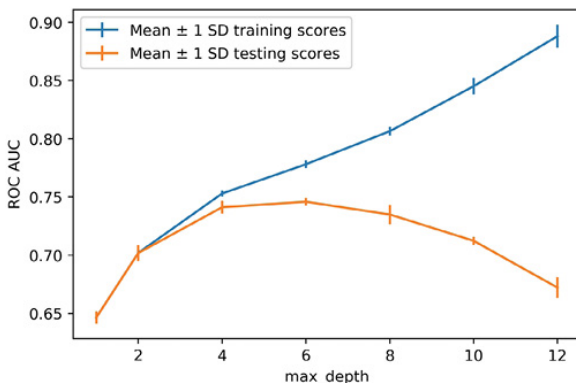


Figura 5.24 – Plotagem de barra de erro dos resultados de treinamento e teste entre os quatro folds.

As barras de erro indicam quanto os resultados variam entre os folds. Se houver muita variação, isso indica que a natureza dos dados entre os folds é diferente de uma maneira que afeta a habilidade de o modelo descrevê-la. Esse problema pode ser preocupante porque implica que podemos não ter dados suficientes para treinar um modelo que apresente um desempenho confiável para novos dados. No entanto, em nosso caso, não há muita variabilidade entre os folds, logo, estamos bem.

E quanto às tendências gerais dos resultados de treinamento e teste entre os diferentes valores de `max_depth`? Podemos ver que, à medida que criamos árvores cada vez mais profundas, o modelo ajusta melhor os dados de treinamento. Como mencionado anteriormente, se criarmos árvores suficientemente grandes para que cada nó folha tenha apenas uma amostra de treinamento, geraremos um modelo muito específico dos dados de treinamento. Na verdade, ele ajustaria os dados de treinamento perfeitamente. Poderíamos dizer que esse seria um modelo de **variância** extremamente alta.

Porém, esse desempenho com o conjunto de treinamento não é necessariamente passado para o conjunto de teste. Na *Figura 5.24*, fica claro que o aumento de `max_depth` só aumenta os resultados do teste até um ponto, depois do qual na verdade árvores mais profundas têm desempenho mais baixo no teste. Esse é outro exemplo de como podemos nos beneficiar do **trade-off entre viés e variância** para criar um modelo preditivo melhor – semelhante a como usamos uma regressão logística regularizada. Árvores menos profundas têm mais **viés**, já que não ajustam tão bem os dados de treinamento. Contudo, isso é bom porque, se aceitarmos algum viés, teremos melhor desempenho nos dados de teste, que é a métrica definitiva usada na

seleção de hiperparâmetros de modelos.

Nesse caso, selecionaríamos `max_depth = 6`. Você também poderia fazer uma pesquisa mais completa, testando todos os inteiros entre 2 e 12, em vez de dar saltos de 2 como fizemos aqui. Geralmente, é uma boa ideia executar uma pesquisa o mais completa possível no espaço dos parâmetros, até os limites do tempo de computação que você tiver. No nosso caso, isso levaria ao mesmo resultado.

Comparação entre modelos

Calculamos a validação cruzada de 4 folds com vários modelos de machine learning usando os dados do estudo de caso. Como estamos nos saindo? Qual é nossa melhor opção até agora? No último capítulo, obtivemos uma ROC AUC de média de teste de 0.718 com a regressão logística e 0.740 criando características de interação por engenharia também em uma regressão logística. Aqui, com uma árvore de decisão, obtivemos 0.745. Logo, estamos ganhando melhorias no desempenho do modelo. Examinaremos mais um tipo de modelo para ver se podemos aumentar ainda mais o desempenho.

Florestas aleatórias: Combinações de árvores de decisão

Como vimos no exercício anterior, as árvores de decisão são propensas ao overfitting. Essa é uma das principais críticas feitas ao seu uso, apesar de elas serem altamente interpretáveis. No entanto, conseguimos limitar esse overfitting, até certo ponto, restringindo a profundidade máxima até a qual a árvore poderia crescer.

Há modelos preditivos poderosos e amplamente usados que empregam árvores de decisão como base para procedimentos mais complexos. Especificamente, aqui daremos ênfase às florestas aleatórias de árvores de decisão. As florestas aleatórias são exemplos do que chamamos de combinação de modelos, porque elas são formadas pelo agrupamento de outros modelos. Com a combinação das previsões de muitos modelos, é possível melhorar as deficiências de cada um deles.

Uma vez entendidas as árvores de decisão, é fácil entender o conceito de florestas aleatórias. Isso é verdade porque as florestas aleatórias são apenas combinações de muitas árvores de decisão; todos os modelos desse tipo de combinação têm a mesma forma matemática. Quantos modelos de árvore de decisão são incluídos em uma floresta aleatória?

Esse é um dos hiperparâmetros, `n_estimators`, que têm de ser especificados na construção de um modelo de floresta aleatória. Geralmente, quanto mais árvores, melhor. À medida que o número de árvores aumenta, a variância do conjunto geral diminui. Isso deve resultar no modelo de floresta aleatória tendo uma maior facilidade para ser generalizado para novos dados, o que será refletido no aumento dos resultados do teste. Porém, haverá um ponto em que o retorno diminuirá após o qual o aumento do número de árvores não resultará em uma melhoria substancial no desempenho do modelo.

Como as florestas aleatórias reduzem o problema de alta variância (overfitting) que afeta as árvores de decisão? A resposta para essa pergunta está no que há de diferente entre as árvores da floresta. Há duas maneiras básicas de as árvores serem diferentes, uma das quais já conhecemos:

- Número de características consideradas em cada divisão
- Amostras usadas na criação de diferentes árvores

Número de características consideradas em cada divisão

Já estamos familiarizados com a opção `max_features` da classe `DecisionTreeClassifier`. Em nosso uso anterior dessa classe, deixamos `max_features` com seu valor padrão `None`, que significa que todas as características serão consideradas em cada divisão. Com o uso de todas as características no ajuste dos dados de treinamento, pode ocorrer overfitting. Se limitarmos o número de características consideradas em cada divisão, algumas das árvores de decisão de uma floresta aleatória encontrarão melhores divisões. Isso ocorre porque, embora elas ainda estejam procurando a melhor divisão, estão fazendo-o com uma seleção de características limitada. Isso pode possibilitar que certas divisões, que talvez não sejam encontradas se todas as características estiverem sendo pesquisadas a cada divisão, sejam feitas posteriormente na árvore.

Há uma opção `max_features` na classe `RandomForestClassifier` do `scikit-learn` que também existe para a classe `DecisionTreeClassifier`, e elas são semelhantes. No entanto, para a floresta aleatória, a configuração padrão é `'auto'`, que significa que o algoritmo fará uma seleção aleatória da raiz quadrada do número de características possíveis em cada divisão, por exemplo, uma seleção aleatória de $\sqrt{9} = 3$ características de um total de 9 características possíveis. Já que provavelmente as árvores da floresta farão seleções aleatórias

diferentes para a divisão quando estiverem sendo estendidas, elas não serão iguais.

Amostras usadas na criação de diferentes árvores

O outro fator que torna as árvores de uma floresta aleatória diferentes é serem criadas com amostras de treinamento distintas. Para que isso ocorra, é usado um procedimento estatístico conhecido como bootstrapping, que significa gerar novos datasets sintéticos a partir dos dados originais. Os datasets sintéticos são criados pela seleção aleatória de amostras no dataset original com o uso da substituição. Aqui, “substituição” significa que, se selecionarmos uma amostra específica, continuaremos a considerá-la para seleção, isto é, ela será “substituída” no dataset original após ser selecionada. O número de amostras dos datasets sintéticos é igual ao do dataset original, mas algumas podem ser repetidas devido à substituição.

O procedimento de usar uma amostragem aleatória para criar datasets sintéticos e treinar modelos com eles separadamente chama-se bagging, que é a abreviação de bootstrapped aggregation. O bagging pode ser usado com qualquer modelo de machine learning, e não apenas com árvores de decisão, e o scikit-learn oferece uma funcionalidade que nos permite usá-lo para problemas tanto de classificação (`BaggingClassifier`) quanto de regressão (`BaggingRegressor`). No caso da floresta aleatória, o bagging é ativado por padrão e a opção de bootstrap é configurada com `True`. No entanto, se você quiser que todas as árvores da floresta sejam criadas usando todos os dados de treinamento, pode configurar essa opção com `False`.

Você já deve ter uma boa ideia do que é uma floresta aleatória. Como pode ver, se estiver familiarizado com as árvores de decisão, entender as florestas aleatórias não envolve muito conhecimento adicional. Algo que confirma esse fato é que os hiperparâmetros disponíveis para a classe `RandomForestClassifier` no scikit-learn são quase iguais aos da classe `DecisionTreeClassifier`.

Além de `n_estimators` e `bootstrap`, que discutimos anteriormente, apenas mais duas opções novas foram acrescentadas às que já estão disponíveis para as árvores de decisão:

- **`oob_score`, um booleano:** Essa opção controla se calcularemos ou não um resultado fora do bag (OOB, out of bag) para cada árvore. Podemos considerá-lo como um resultado de teste, em que as

amostras que não foram selecionadas pelo procedimento de bagging para estender uma árvore específica são usadas na avaliação do desempenho do modelo dessa árvore. Usamos True para calcular o resultado OOB ou False (o padrão) para não o fazer.

- **warm_start, um booleano:** Essa opção é False por padrão – se você configurá-la com True, reutilizar o mesmo objeto de modelo de floresta aleatória fará com que árvores adicionais sejam adicionadas à floresta já gerada.

Outros tipos de combinações de modelos

A floresta aleatória, como sabemos agora, é um exemplo de combinação de bagging. Outro tipo de combinação é a de **boosting**. A ideia geral que envolve o boosting é o uso de sucessivos modelos novos do mesmo tipo e seu treinamento baseado nos erros de modelos anteriores. Dessa forma, os modelos sucessivos aprendem onde os modelos anteriores não se saíram bem e corrigem esses erros. O boosting tem sido aplicado exitosamente com árvores de decisão e está disponível no scikit-learn e em outro popular pacote Python chamado XGBoost.

O **stacking** é uma combinação um pouco mais avançada, em que os diferentes modelos (estimadores) não precisam ser do mesmo tipo como o são no bagging e boosting. Por exemplo, você poderia construir uma combinação de stacking com uma floresta aleatória e uma regressão logística. As previsões dos diferentes membros da combinação são reunidas em uma previsão final com o uso de mais um modelo (o **empilhador**), que considera as previsões dos modelos **empilhados** como características.

Floresta aleatória: Previsões e interpretabilidade

Já que uma floresta aleatória é apenas um conjunto de árvores de decisão, as previsões de todas essas árvores devem ser combinadas de alguma forma para criar sua previsão.

Após o treinamento do modelo, as árvores de classificação receberão uma amostra como entrada e produzirão uma classe prevista, por exemplo, se ou não uma conta de crédito do problema de nosso estudo de caso ficará inadimplente. Uma abordagem intuitiva para a combinação das previsões dessas árvores na previsão final da floresta é o uso de um voto majoritário. Isto é, a previsão mais comum entre todas as árvores será a previsão da floresta. Na verdade, essa foi a

abordagem usada na publicação que descreveu pela primeira vez as florestas aleatórias (<https://scikit-learn.org/stable/modules/ensemble.html#forest>). No entanto, o scikit-learn usa uma abordagem um pouco diferente: a soma das probabilidades previstas para cada classe e a seleção da de maior probabilidade. Esse método captura mais informações de cada árvore, e não apenas a classe prevista.

Interpretabilidade das florestas aleatórias

Uma das principais vantagens das árvores de decisão é que é fácil ver como uma previsão individual é feita. Você pode rastrear o caminho da decisão para qualquer amostra prevista por meio da série de regras “se então” usadas para fazer a previsão e saber exatamente como chegamos a ela. Por outro lado, suponhamos que tivéssemos uma floresta aleatória composta de 1.000 árvores. Isso significaria que há 1.000 conjuntos de regras como essa, o que é muito mais difícil de comunicar para os seres humanos do que um único conjunto de regras!

Dito isso, há vários métodos que podem ser usados para entendermos como as florestas aleatórias fazem previsões. Há técnicas avançadas para a compreensão de como as previsões individuais são feitas, que são o assunto de uma pesquisa acadêmica recente e não fazem parte do escopo deste livro. No entanto, uma maneira simples de interpretar o modo como uma floresta aleatória funciona, que está disponível no scikit-learn, é observar a importância das características. A importância das características de uma floresta aleatória mede quanto cada característica foi útil no crescimento das árvores da floresta. Sua utilidade é medida com o uso da combinação da fração de amostras de treinamento que foram divididas empregando essa característica e da diminuição obtida na impureza do nó.

Devido ao cálculo da importância das características, que pode ser usado na classificação de características de acordo com sua utilidade para o modelo de floresta aleatória, as florestas aleatórias também podem ser empregadas na seleção de características.

Exercício 21: Ajustando uma floresta aleatória

Neste exercício, estenderemos os esforços que fizemos com as árvores de decisão, usando o modelo de floresta aleatória com a validação cruzada para os dados de treinamento do estudo de caso. Observaremos o efeito do aumento do número de árvores na floresta e

examinaremos a importância das características que pode ser calculada com o uso de um modelo de floresta aleatória. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado em <https://bit.ly/2UpGDW2>.

1. Importe o modelo classificador de floresta aleatória como mostrado a seguir:

```
from sklearn.ensemble import RandomForestClassifier
```

2. Instancie a classe usando estas opções:

```
rf = RandomForestClassifier(  
    n_estimators=10, criterion='gini', max_depth=3,  
    min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,  
    max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,  
    min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,  
    random_state=4, verbose=0, warm_start=False, class_weight=None)
```

Para este exercício, usaremos principalmente as opções padrão. Porém, observe que configuramos `max_depth = 3`. Aqui, só examinaremos o efeito do uso de diferentes números de árvores, que ilustraremos com árvores relativamente rasas para obter runtimes mais curtos. Para encontrar o melhor desempenho do modelo, normalmente faríamos testes com profundidades maiores e em maior número para as árvores.

Também configuramos `random_state` para obter resultados previsíveis entre as execuções.

3. Crie uma grade de parâmetros para este exercício para pesquisar os números de árvores variando de 10 a 100:

```
rf_params_ex = {'n_estimators':list(range(10,110,10))}
```

Usamos a função `range()` de Python para criar um iterador para os valores inteiros que queremos e depois os convertimos em uma list usando `list()`.

4. Instancie um objeto de validação cruzada por busca em grade para o modelo de floresta aleatória usando a grade de parâmetros da etapa anterior. Caso contrário, você pode usar as mesmas opções de validação cruzada da árvore de decisão:

```
cv_rf_ex = GridSearchCV(rf, param_grid=rf_params_ex, scoring='roc_auc',  
    fit_params=None, n_jobs=None, iid=False, refit=True,  
    cv=4, verbose=1, pre_dispatch=None, error_score=np.nan,  
    return_train_score=True)
```

5. Ajuste o objeto de validação cruzada como mostrado a seguir:

```
cv_rf_ex.fit(X_train, y_train)
```

O procedimento de ajuste deve exibir o seguinte:

```
Fitting 4 folds for each of 10 candidates, totalling 40 fits
```

```
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.  
[Parallel(n_jobs=1)]: Done 40 out of 40 | elapsed: 22.9s finished
```

```
GridSearchCV(cv=4, error_score=nan,  
             estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
                                               max_depth=3, max_features='auto', max_leaf_nodes=None,  
                                               min_impurity_decrease=0.0, min_impurity_split=None,  
                                               min_samples_leaf=1, min_samples_split=2,  
                                               min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,  
                                               oob_score=False, random_state=4, verbose=0, warm_start=False),  
             fit_params=None, iid=False, n_jobs=None,  
             param_grid={'n_estimators': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]},  
             pre_dispatch=None, refit=True, return_train_score=True,  
             scoring='roc_auc', verbose=1)
```

Figura 5.25 – Saída obtida na validação cruzada da floresta aleatória para diferentes números de árvores.

Você deve ter notado que, embora só estejamos usando a validação cruzada para 10 valores de hiperparâmetros, em comparação com os 7 valores que examinamos para a árvore de decisão do exercício anterior, essa validação cruzada demorou mais. Considere quantas árvores estamos criando nesse caso. Para o último hiperparâmetro, `n_estimators = 100`, criamos um total de 400 árvores para todas as divisões da validação cruzada.

Quanto tempo demorou o ajuste do modelo para os diferentes números de árvores que acabamos de usar? Que ganhos foram gerados para o desempenho do teste na validação cruzada com o uso de mais árvores? Essas são questões interessantes para serem examinadas com o uso de plotagens. Primeiro, passaremos os resultados da validação cruzada para um DataFrame do pandas como fizemos antes.

6. Insira os resultados da validação cruzada em um DataFrame do pandas:

```
cv_rf_ex_results_df = pd.DataFrame(cv_rf_ex.cv_results_)
```

Você pode examinar o DataFrame inteiro no Jupyter notebook. Aqui, passaremos diretamente à criação das plotagens dos valores de interesse. Criaremos uma plotagem linear, com símbolos, do tempo de ajuste médio entre os folds para cada hiperparâmetro, contido na coluna `mean_fit_time`, assim como uma plotagem de barra de erro dos resultados do teste, o que já fizemos para as árvores de decisão. As duas plotagens serão criadas em relação a `max_depth` no eixo x.

7. Crie duas subplotagens, do tempo médio e dos resultados de média do teste com desvio padrão:

```
fig, axs = plt.subplots(nrows=1, ncols=2, figsize=(6, 3))
axs[0].plot(cv_rf_ex_results_df['param_n_estimators'],
            cv_rf_ex_results_df['mean_fit_time'],
            '-o')
axs[0].set_xlabel('Number of trees')
axs[0].set_ylabel('Mean fit time (seconds)')
axs[1].errorbar(cv_rf_ex_results_df['param_n_estimators'],
               cv_rf_ex_results_df['mean_test_score'],
               yerr=cv_rf_ex_results_df['std_test_score'])
axs[1].set_xlabel('Number of trees')
axs[1].set_ylabel('Mean testing ROC AUC  $\pm$  1 SD ')
plt.tight_layout()
```

Aqui, usamos `plt.subplots` para criar dois eixos ao mesmo tempo dentro de uma figura em uma configuração de uma linha por duas colunas. Em seguida, acessamos os objetos de eixo, indexando o array de eixos `axs` retornado por essa operação para criar plotagens.

A saída deve ser esta plotagem:

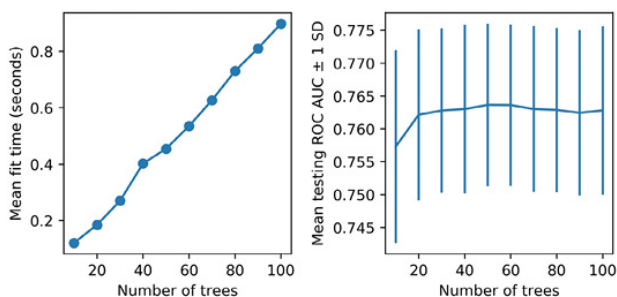


Figura 5.26 – O tempo de ajuste e os resultados de teste médios de diferentes números de árvores da floresta.

Nota: Seus resultados podem não ser iguais devido a diferenças na plataforma ou se você definir um seed aleatório diferente.

Há várias coisas que podemos observar nessas visualizações. Em primeiro lugar, podemos ver que, usando uma floresta aleatória, o desempenho do modelo nos folds de teste da validação cruzada foi melhor que o de qualquer um de nossos esforços anteriores. Embora não tenhamos tentado ajustar os hiperparâmetros da floresta aleatória para chegar ao melhor desempenho possível, esse é um resultado promissor e indica que uma floresta aleatória será um acréscimo valioso aos nossos esforços de modelagem.

No entanto, junto com esses resultados mais altos para o teste do

modelo, observe que também há mais variabilidade entre os folds do que vimos com a árvore de decisão; essa variabilidade é vista como desvios padrão maiores nos resultados de teste do modelo entre os folds. Embora isso indique que há uma variação no desempenho maior do que a esperada com o uso desse modelo, recomendamos que você examine os resultados de teste dos folds diretamente no DataFrame do pandas no Jupyter notebook. Deve ver que até mesmo o menor resultado de um fold individual ainda é maior que o resultado médio de teste da árvore de decisão, indicando que será melhor usar uma floresta aleatória.

E quanto às outras questões que queríamos examinar com essa visualização? Queremos ver quanto leva o ajuste de modelos de floresta aleatória com vários números de árvores e quais são os ganhos no desempenho do modelo com o uso de mais árvores. A subplotagem da esquerda na *Figura 5.26* mostra que há um aumento linear no tempo de treinamento à medida que mais árvores são adicionadas à floresta. Isso é esperado; estamos multiplicando o volume de computação a ser executado no procedimento de treinamento com a inclusão de mais árvores.

Esse tempo de computação adicional vale a pena em termos de aumento de desempenho do modelo? A subplotagem da direita na *Figura 5.26* mostra que acima de cerca de 20 árvores não fica claro se a inclusão de mais árvores melhora realmente o desempenho do teste. Embora o modelo com 50 árvores tenha o resultado mais alto, o fato de a inclusão de mais árvores na verdade diminuir um pouco a pontuação de teste indica que o ganho na ROC AUC para 50 árvores deve ter ocorrido devido à aleatoriedade, já que teoricamente incluir mais árvores deve aumentar o desempenho do modelo. De acordo com esse raciocínio, se estivéssemos limitados a `max_depth = 3`, poderíamos optar por uma floresta de 20 ou talvez 50 árvores e prosseguir. Contudo, examinaremos o espaço dos parâmetros mais detalhadamente na atividade do fim deste capítulo.

Para concluir, observe que não mostramos as métricas ROC AUC de treinamento. Se você as plotasse ou as procurasse no DataFrame de resultados, veria que as pontuações de treinamento são maiores que as de teste, indicando que algum nível de overfitting está ocorrendo. Embora possa ser esse o caso, também é verdade que a pontuação de teste desse modelo de floresta aleatória é mais alta do que a que observamos para qualquer outro modelo. Com base nesse

resultado, provavelmente selecionaríamos o modelo de floresta aleatória.

Como aprendizado adicional sobre o que podemos acessar usando nosso objeto de validação cruzada ajustado, examinaremos os melhores hiperparâmetros e a importância das características.

8. Use este código para ver os melhores hiperparâmetros fornecidos pela validação cruzada:

```
cv_rf_ex.best_params_
```

O código deve exibir:

```
{'n_estimators': 50}
```

Figura 5.27 – Melhores hiperparâmetros da validação cruzada.

Aqui, “melhor” significa apenas os hiperparâmetros que resultaram na mais alta pontuação média de teste do modelo.

9. Execute este código para criar um DataFrame dos nomes e da importância das características e exiba-o classificado por importância:

```
feat_imp_df = pd.DataFrame({
    'Feature name': features_response[:-1],
    'Importance': cv_rf_ex.best_estimator_.feature_importances_
})
feat_imp_df.sort_values('Importance', ascending=False)
```

As primeiras linhas da saída devem ser as seguintes:

	Feature name	Importance
4	PAY_1	0.609609
11	PAY_AMT1	0.094123
0	LIMIT_BAL	0.079265
13	PAY_AMT3	0.047067
12	PAY_AMT2	0.035393

Figura 5.28 – Importância das características de uma floresta aleatória.

Nesse código, criamos um dicionário com os nomes e a importância das características. A importância das características veio do método `best_estimator_` do objeto de validação cruzada ajustado. Essa é uma maneira de acessarmos o objeto de modelo de floresta aleatória que foi treinado com todos os dados de treinamento usando os melhores hiperparâmetros que visualizamos na etapa anterior. `feature_importances_` é um método que pode ser usado em modelos de floresta aleatória ajustados. Após inserir os nomes e a importância das características em um dicionário, criamos o

DataFrame e depois o exibimos classificado na ordem decrescente por importância. Observe que as 5 características mais importantes da floresta aleatória são as mesmas selecionadas por um teste F ANOVA no *Capítulo 3, Detalhes da regressão logística e exploração de características*, embora estejam em uma ordem um pouco diferente. Essa é uma boa confirmação entre os diferentes métodos.

Gráfico quadriculado (checkerboard)

Antes de passar para a Atividade, ilustraremos uma técnica de visualização do Matplotlib. A plotagem de uma grade bidimensional com quadrados coloridos ou outras formas pode ser útil quando você quiser exibir três dimensões de dados. A cor ilustra a terceira dimensão. Por exemplo, você pode querer visualizar as pontuações de teste do modelo em uma grade de dois hiperparâmetros. Ou seja, na verdade estamos falando do caso de uso da *Atividade 5, Busca em grade na validação cruzada com floresta aleatória*.

A primeira etapa do processo é a criação de grades de coordenadas x e y. A função `meshgrid` do NumPy pode ser usada para isso. Ela recebe arrays unidimensionais de coordenadas x e y e cria a grade de malha com todos os pares de coordenadas possíveis. Os pontos da grade de malha serão os cantos da plotagem quadriculada. É assim que o código encontra uma grade 4 x 4 de quadrados coloridos. Já que estamos especificando os cantos, precisamos de uma grade 5 x 5:

```
xx_example, yy_example = np.meshgrid(range(5), range(5))
print(xx_example)
print(yy_example)
```

A saída é:

```
[[0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]]
[[0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]]
```

Figura 5.29 – Saída da grade de malha.

A grade de dados que será plotada nessa malha também deve ter uma forma quadrada. Pegaremos um array unidimensional de inteiros entre 1 e 16 e o reconstruiremos como uma grade bidimensional 4 x 4:

```
z_example = np.arange(1,17).reshape(4,4)
```

z_example

```
array([[ 1,  2,  3,  4],  
       [ 5,  6,  7,  8],  
       [ 9, 10, 11, 12],  
       [13, 14, 15, 16]])
```

Figura 5.30 – Dados da plotagem quadriculada.

Podemos plotar os dados de `z_example` na grade de malha `xx_example`, `yy_example` com o código a seguir. Observe que usamos `pcolormesh` para criar a plotagem com o mapa de cores `jet`, que fornece a escala de cores do arco-íris. Adicionamos um colorbar, que tem de receber o objeto `pcolor_ex` retornado por `pcolormesh` como argumento para que a interpretação da escala de cores fique clara:

```
ax = plt.axes()  
pcolor_ex = ax.pcolormesh(xx_example, yy_example, z_example, cmap=plt.cm.jet)  
plt.colorbar(pcolor_ex, label='Color scale')  
ax.set_xlabel('X coordinate')  
ax.set_ylabel('Y coordinate')
```

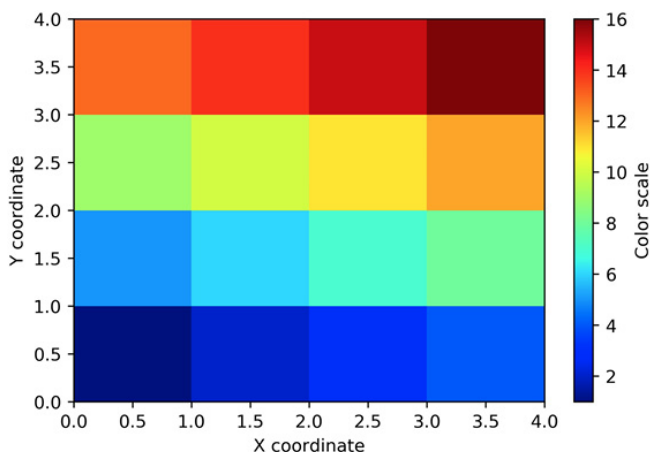


Figura 5.31 – Plotagem de inteiros consecutivos com pcolormesh.

Atividade 5: Busca em grade na validação cruzada com floresta aleatória

Nesta atividade, você conduzirá uma busca em grade pelas diversas árvores da floresta (`n_estimators`) e até a profundidade máxima de uma árvore (`max_depth`) do modelo de floresta aleatória com os dados do estudo de caso. Em seguida, criará uma visualização exibindo o resultado da média de teste para a grade de hiperparâmetros que pesquisou. Execute as etapas a seguir para concluir a atividade:

Nota: O código e a saída resultante desta atividade foram carregados em um Jupyter Notebook que pode ser encontrado em <http://bit.ly/2GI7fJB>.

1. Crie um dicionário representando a grade para os hiperparâmetros `max_depth` e `n_estimators` que serão pesquisados. Inclua as profundidades 3, 6, 9 e 12, e 10, 50, 100 e 200 árvores. Deixe os outros hiperparâmetros com seus padrões.
2. Instancie um objeto `GridSearchCV` usando as mesmas opções que empregamos anteriormente neste capítulo, mas com o dicionário de hiperparâmetros criado na etapa 1 desta atividade. Configure `verbose=2` para ver a saída de cada ajuste executado. Você pode reutilizar o mesmo objeto de modelo de floresta aleatória `rf` que estivemos usando.
3. Ajuste o objeto `GridSearchCV` com os dados de treinamento.
4. Insira os resultados da busca em grade em um `DataFrame` do `pandas`.
5. Crie uma visualização com `pcolormesh` para o resultado da média de teste de cada combinação de hiperparâmetros.
6. Decida que conjunto de hiperparâmetros usará.

Nota

A solução desta atividade pode ser encontrada na página XXX.

Resumo

Neste capítulo, aprendemos como usar as árvores de decisão e as combinações de modelos chamadas florestas aleatórias que são compostas de várias árvores de decisão. Usando esses modelos concebidos de maneira simples, pudemos criar previsões melhores do que as que criaríamos com a regressão logística, se avaliarmos pelo resultado da ROC AUC da validação cruzada. É isso que costuma ocorrer em muitos problemas do mundo real. As árvores de decisão são robustas para vários problemas que podem impedir os modelos de regressão logística de ter um bom desempenho, como os relacionamentos não lineares entre as características e a variável de resposta, e a presença de interações complicadas entre as características.

Embora uma única árvore de decisão esteja propensa ao overfitting, o

método de combinação da floresta aleatória tem comprovado que reduz esse problema de alta variância. As florestas aleatórias são construídas pelo treinamento de muitas árvores. A diminuição da variância na combinação das árvores é obtida pelo aumento do viés das árvores individuais da floresta, simplesmente pelo seu treinamento com uma parte do conjunto de treinamento disponível (bootstrapped aggregation ou bagging) e considerando apenas um número reduzido de características a cada divisão de nó.

Já testamos várias abordagens de machine learning para modelar os dados do estudo de caso. Vimos que algumas funcionam melhor do que outras; por exemplo, uma floresta aleatória com hiperparâmetros ajustados fornece o resultado médio mais alto de 0.776 para a ROC AUC da validação cruzada.

Porém, no fim das contas, o resultado da ROC AUC é uma métrica abstrata de avaliação de um modelo. O cliente pode não ter um conhecimento suficiente do contexto para saber o que o resultado da ROC AUC significa na prática, mas ele precisa saber se o modelo pode atender às suas necessidades empresariais. Portanto, uma etapa final crucial do machine learning em um contexto empresarial é tentar determinar o valor financeiro, ou operacional, de um modelo preditivo. Mostraremos como fazer isso, além de abordar outras questões relacionadas à distribuição dos modelos que serão usados para guiar decisões empresariais, no próximo capítulo.

Imputação de dados faltantes, análise financeira e distribuição para o cliente

Objetivos do aprendizado

Ao fim deste capítulo, você conseguirá:

- Comparar os resultados de todos os modelos construídos para um estudo de caso
- Substituir dados faltantes usando um conjunto de estratégias de imputação
- Construir um modelo de classificação multiclasse
- Conduzir uma análise financeira para encontrar o limite ótimo da classificação binária
- Obter insights financeiros a partir do modelo para ajudar o cliente a conduzir a estratégia orçamentária e operacional
- Distribuir o modelo e fazer recomendações de uso

Este capítulo apresentará uma comparação dos resultados dos diversos modelos construídos para um estudo, descreverá os insights financeiros derivados do modelo final e resumirá as últimas etapas do projeto necessárias para atendermos aos requisitos do cliente.

Introdução

No capítulo anterior, introduzimos as árvores de decisão e as florestas aleatórias e vimos como elas podem ser usadas para melhorar a qualidade da modelagem preditiva dos dados do estudo de caso.

Neste capítulo, consideraremos a construção de modelos como

concluída e abordaremos todas as outras questões sobre as quais precisamos refletir antes de distribuir o modelo para o cliente. Os dois elementos-chave do capítulo são a imputação de dados e a análise financeira.

Com a imputação de dados, examinaremos várias estratégias de como fazer suposições embasadas para os valores faltantes de características do dataset. Isso lhe permitirá fazer previsões para todas as amostras.

Na análise financeira, você dará os passos finais, porém cruciais, para entender como um modelo pode ser usado no mundo real. Provavelmente seu cliente apreciará seus esforços de criação de um modelo mais preciso ou com uma ROC AUC mais alta. No entanto, o que lhe agradará mais é saber quanto o modelo pode ajudá-lo a ganhar ou economizar financeiramente e ficará feliz em receber uma orientação específica sobre como maximizar o potencial do modelo nessa questão. Após examinar a imputação e a análise financeira, terminaremos com algumas considerações sobre como distribuir e monitorar o modelo.

Revisão dos resultados dos modelos

Para desenvolver um modelo de classificação binária que atenda aos requisitos empresariais de nosso cliente, testamos várias abordagens de modelagem com diferentes níveis de sucesso. No término do trabalho, selecionaremos a que funcionou melhor, para executar análises adicionais e apresentar para o cliente. Porém, também é bom mostrar para o cliente as descobertas que fizemos nas várias opções que exploramos. Isso mostrará que fizemos um trabalho completo.

Revisaremos os diferentes modelos que testamos para o problema do estudo de caso, os hiperparâmetros que tivemos de ajustar e os resultados da validação cruzada. Só incluiremos o trabalho que fizemos usando todas as características possíveis, e não os modelos anteriores em que usamos apenas uma ou duas características (por exemplo, EDUCATION) como uma maneira de aprender como empregar funções de ajuste de modelos no scikit-learn.

ROC AUC média da validação cruzada para diferentes conjuntos de treinamento			
Figura 15-6: Validando o modelo de classificação binária com os dados do estudo de caso			
Figura 15-7: Validando o modelo de classificação binária com as engenharias de interação criadas com o banco de dados do estudo de caso			
Figura 15-8: Encontrando hiperparâmetros ótimos para uma árvore de decisão			
Figura 15-9: Validando o modelo de classificação binária com a validação cruzada com floresta aleatória			

Figura 6.1 – Resumo das atividades de modelagem com os dados do estudo de caso.

Na *Figura 6.1*, podemos ver que, para esse problema específico, nossos esforços de criação de **modelos mais complexos**, gerando novas características por engenharia para adicioná-las a uma regressão logística simples ou criando uma combinação de árvores de decisão, produziram melhoria no desempenho do modelo.

Na apresentação dos resultados para o cliente, seria melhor mostrar apenas a primeira e a última colunas da *Figura 6.1*. Provavelmente ele não estará interessado em todos os detalhes da coluna “Hiperparâmetros ajustados”, a menos que tenha conhecimento técnico. Você deve estar preparado para interpretar os resultados para os sócios da empresa em todos os níveis de familiaridade técnica, inclusive para os que tiverem pouca experiência prática.

Uma situação comum é o cliente não saber de onde vem a medida de ROC AUC. Nesse caso, você teria de explicar que se trata de uma métrica que pode variar entre 0.5 e 1 e fazer explicações intuitivas sobre esses limites: 0.5 é como se jogássemos uma moeda para o alto e 1 é o ideal, que não é atingível. Nossos resultados ficaram em um espaço intermediário, chegando perto de 0.8 com o melhor modelo que desenvolvemos. Embora esse número sozinho não seja necessariamente significativo, levado em consideração com todos os modelos que testamos e as limitações da medida de ROC AUC, ele mostra que usamos vários métodos e conseguimos melhorar o desempenho. Por fim, para uma aplicação financeira como a do estudo de caso, métricas abstratas de desempenho do modelo devem ser acompanhadas por uma análise financeira se possível. Examinaremos isso posteriormente neste capítulo.

Dependendo do tempo que você tiver para executar um projeto e de sua experiência em diferentes técnicas de modelagem, deve testar o maior número de métodos de modelagem que puder. Métodos mais avançados, como as árvores de decisão de aumento de gradiente ou as redes neurais para classificação, podem produzir um desempenho melhor para esse problema. Recomendamos que você continue seus estudos e aprenda como usar esses modelos. Especificamente, agora que você conhece as árvores de decisão e as florestas aleatórias, as árvores de aumento de gradiente seriam uma próxima etapa lógica, já que são uma combinação de modelos baseada nas árvores de decisão, semelhante às florestas aleatórias. No entanto, para o estudo de caso, assumiremos que chegamos ao final do prazo que planejamos para a

modelagem do modelo. Agora, temos de passar para as preocupações práticas associadas à distribuição de um modelo para nosso cliente.

Nossa conclusão é a de que um modelo de floresta aleatória, com os hiperparâmetros que determinamos na validação cruzada (200 árvores e profundidade máxima de 9), é o modelo que recomendamos.

Como observação final sobre a construção de modelos para o estudo de caso, observe que não abordamos o desbalanceamento de classes na variável de resposta. Tente ajustar os modelos com a opção `class_weight='balanced'` do scikit-learn para ver o efeito; você verá que ela não melhorará substancialmente o desempenho do modelo, já que o desbalanceamento de classes nos dados do estudo de caso não é particularmente severo.

Lidando com dados faltantes: Estratégias de imputação

Você deve se lembrar de que, no *Capítulo 1, Exploração e limpeza de dados*, recomendamos o uso de uma proporção de amostras relativamente grande no dataset ($3,021/29,685 = 10.2\%$) na qual o valor da característica PAY_1 estava faltando. Esse é um problema que tem de ser resolvido, porque muitos algoritmos de machine learning, inclusive as implementações de regressão logística e floresta aleatória do scikit-learn, não aceitam entradas para o treinamento ou o teste do modelo que tenham valores faltando.

Nossa solução para esse problema foi simplesmente descartar todas as amostras que tinham valores faltando para PAY_1. No entanto, após discutir a questão com nosso cliente, soubemos que os valores estavam faltando devido a um problema de relatório que eles estavam tentando corrigir. A curto prazo, se houver um método disponível que permita a inclusão das contas com informações faltantes de PAY_1 no processo de previsão do modelo, isso é preferível. Logo, temos de considerar como fazer previsões para contas em que essa característica esteja faltando.

Resolver o problema de dados faltantes é uma habilidade básica para os cientistas de dados, já que isso é algo que ocorre muito no mundo real, datasets “sujos”. Em vez de ignorar as amostras com dados faltantes, outra opção é usar algum método para fornecer os valores ausentes de características afetadas. Esse fornecimento é conhecido como **imputação**. Os métodos para o trabalho com dados faltantes

variam de simples a mais complexos, como mostrado na *Figura 6.2*.

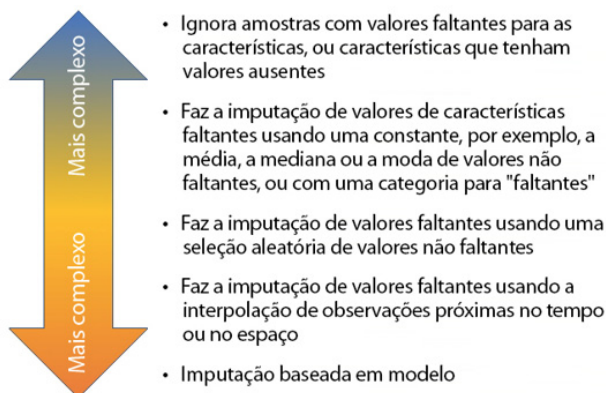


Figura 6.2 – Estratégias para a imputação de valores faltantes de características.

A maneira mais simples de lidar com amostras que tenham dados faltando para certas características é remover essas amostras do dataset ou remover as características que tenham dados ausentes. Embora essa abordagem nos permita prosseguir imediatamente, ela produz o efeito indesejável de descartarmos dados potencialmente úteis do dataset. No caso dos dados do estudo de caso, você sabe que teremos de criar previsões para as amostras mesmo se os valores das características estiverem faltando, logo, temos de incluir esses dados de alguma forma.

Como alternativa ao descarte de dados devido a valores faltantes de características, você pode tentar empregar algum tipo de estratégia de imputação. A imputação envolve usar os valores conhecidos de uma característica específica para fazer uma suposição bem embasada de quais seriam os valores faltantes. Os tipos mais simples de imputação envolvem o uso de uma síntese estatística dos valores não faltantes da característica como o único valor **constante** que substituirá todos os valores faltantes. Essa síntese estatística pode ser a média, a mediana ou a moda para características contínuas. Para características categóricas, a moda é uma opção e no caso de características categóricas ordinais podemos usar a mediana.

Um caso adicional importante de imputação com um valor constante para variáveis categóricas é a *criação de um novo nível da variável categórica para indicar que dados estão faltando*. Por exemplo, se a característica EDUCATION tivesse dados faltando, além dos níveis já existentes da característica, poderíamos criar um novo nível indicando

que dados de EDUCATION estão “ausentes”. Essa pode ser a melhor abordagem quando os dados faltantes de uma característica indicarem algo sobre a variável de resposta. Nesse caso, o fato de valores estarem faltando representa informações importantes que não devem ser substituídas por outros valores para uma característica. Para tomar a decisão de que método usar para a imputação com um valor constante, você pode tentar empregar a validação cruzada como faria para a seleção dos hiperparâmetros do modelo.

Métodos de imputação mais sofisticados podem ser usados no preenchimento de valores faltantes com o uso de valores não constantes. Esses métodos refletem o reconhecimento de que os valores faltantes podem não ser iguais. A maneira mais simples de fazer isso é fornecer os valores faltantes usando uma parcela **aleatória do** conjunto de valores não faltantes por meio da substituição. Dessa forma, a frequência relativa dos diferentes valores selecionados para o fornecimento de dados faltantes será semelhante à dos dados existentes para essa característica.

No entanto, o método de imputação aleatória não usa outras informações de uma amostra específica com dados faltantes quando são selecionados os valores de preenchimento. Se as amostras estiverem relacionadas a tempo ou espaço, como uma série temporal ou dados geolocalizados, métodos de **interpolação** temporal ou espacial podem ser usados. Esses métodos seguem a ideia geral de que um ponto de dados ausente provavelmente está localizado em algum local entre os valores de pontos de dados adjacentes existentes no tempo ou espaço. Há um rico conjunto de métodos de interpolação disponíveis no NumPy e no Pandas que você pode examinar se estiver trabalhando com dados faltantes para séries temporais ou dados geolocalizados.

Para concluir, talvez a maneira mais sofisticada de fornecer dados faltantes seja considerar a questão da imputação um “problema dentro de outro problema” de modelagem preditiva. Nesse método, a característica com valores faltantes é tratada como a variável de resposta e as sem valores faltantes são consideradas características do subproblema. Você pode pegar as amostras que têm valores conhecidos para todas as características, dividi-las em conjuntos de treinamento e teste como faria para qualquer outro problema de aprendizado supervisionado e desenvolver um modelo preditivo para a característica com valores faltantes. Assim poderá usar esse modelo para prever os valores desconhecidos da característica. Já que esse

método depende da modelagem preditiva, vamos chamá-lo de **imputação baseada em modelo**. Vamos usá-lo posteriormente com os dados do estudo de caso para mostrar como funciona.

Observe que, para o dataset de nosso estudo de caso, a característica que tem valores faltantes é PAY_1, que é a característica mais importante como identificado tanto pela seleção univariada quanto pela avaliação da importância das características na floresta aleatória. É esperado que a qualidade da estratégia de imputação implementada para essa característica tenha um impacto mais significativo sobre o desempenho do modelo do que se estivéssemos fazendo a imputação para uma característica menos importante. Examinaremos então todo o conjunto de opções de imputação disponíveis.

Preparando amostras com dados faltantes

Para testar nossas diferentes estratégias com os dados do estudo de caso e ver como afetam a capacidade preditiva do tipo de modelagem que usaremos, temos de trazer as amostras com dados faltantes de volta para o dataset. Para fazê-lo, precisamos usar os mesmos procedimentos de limpeza de dados que empregamos no *Capítulo 1, Exploração de limpeza de dados*; vamos executá-los no próximo exercício.

Exercício 22: Limpando o dataset

Neste exercício, limparemos nosso dataset para resolver o problema das entradas de dados faltantes. Usaremos a mesma abordagem do *Capítulo 1, Exploração e limpeza de dados*. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante dos exercícios 22-25 e da Atividade 6 foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2PtdRyj>. Você pode rolar para a seção apropriada dentro do Jupyter Notebook para localizar o exercício ou a atividade que deseja.

1. Carregue o dataset original que usamos no começo de nossa exploração antes de a limpeza ocorrer:

```
df_orig = pd.read_excel('../Data/  
default_of_credit_card_clients_courseware_version_1_21_19.xls')
```

Temos de repetir todas as etapas de limpeza de dados com exceção da remoção das amostras em que a característica PAY_1 tinha

valores faltantes. Como primeira etapa, identifique e remova qualquer amostra em que os valores de todas as características sejam iguais a 0. Determinamos que essa seria uma maneira eficaz de remover IDs de conta duplicados do dataset.

2. Crie um array booleano que indique onde as entradas do DataFrame são iguais a 0:

```
df_zero_mask = df_orig == 0
```

3. Reduza esse array booleano bidimensional para uma única dimensão indicando que linhas têm 0's em todas as colunas, começando pela segunda coluna. Esse array informará que linhas têm valor 0 para todas as características e precisam ser removidas:

```
feature_zero_mask = df_zero_mask.iloc[:,1:].all(axis=1)
```

4. Verifique se o número de linhas em que todos os valores são 0 para as características é igual ao do *Capítulo 1, Exploração e limpeza de dados*:

```
sum(feature_zero_mask)
```

A saída deve ser:

315

Figura 6.3 – Quantas linhas têm valor 0 para todas as características.

O resultado é o mesmo do *Capítulo 1, Exploração e limpeza de dados*.

5. Use esta máscara para selecionar as outras linhas, isto é, as que não têm valores 0 para todas as características, e verifique a dimensão do DataFrame resultante:

```
df_clean = df_orig.loc[~feature_zero_mask,:].copy()
```

```
df_clean.shape
```

Você deve obter a seguinte saída:

(29685, 25)

Figura 6.4 – Dimensão de um DataFrame limpo.

Além de remover dados claramente inválidos (amostras com todas as características iguais a 0), o objetivo desse procedimento era eliminar IDs de conta duplicados.

6. Verifique se o número de IDs de conta exclusivos é igual ao número de linhas do DataFrame limpo:

```
df_clean['ID'].nunique()
```

A saída deve ser:

Figura 6.5 – Número de IDs de conta exclusivos nos dados limpos.

O resultado é igual ao número de linhas do DataFrame, indicando que os dados foram limpos com a remoção de IDs de conta duplicados.

7. Substitua os valores não documentados das características EDUCATION e MARRIAGE pelo valor documentado para “desconhecido”. Simplesmente repita o código do *Capítulo 1, Exploração e limpeza de dados* que faz isso:

```
df_clean['EDUCATION'].replace(to_replace=[0, 5, 6], value=4, inplace=True)
df_clean['MARRIAGE'].replace(to_replace=0, value=3, inplace=True)
```

Agora os dados estão em um estado limpo. A pergunta que devemos fazer então é como incluir as amostras com valores faltantes para PAY_1 com o resto dos dados.

Reservamos um conjunto de teste desconhecido de todas as atividades de modelagem que executamos até aqui. As amostras desse dataset só devem ser usadas depois que selecionarmos nosso modelo final e quisermos ver se o desempenho fora da amostra que estimamos com a validação cruzada se reflete nos novos dados. Logo, seria melhor continuar mantendo essas amostras isoladas do trabalho de modelagem, inclusive da seleção de uma estratégia de imputação. Para fazê-lo, criaremos um subconjunto das amostras de nosso dataset limpo que têm valores faltantes para PAY_1 e as adicionaremos aos conjuntos de treinamento e teste posteriormente, para que a proporção de dados de treinamento e teste permaneça igual.

8. Recapitule quais são os valores de PAY_1:

```
df_clean['PAY_1'].value_counts()
```

A saída deve ser:

0	13087
-1	5047
1	3261
Not available	3021
-2	2476
2	2378
3	292
4	63
5	23
8	17
6	11
7	9

Figura 6.6 – Valores de PAY_1.

Antes de isolar as amostras em que o valor de PAY_1 está faltando, observe que a moda de valores não faltantes de PAY_1, ou seja, o valor predominante, é 0. Usaremos esse fato posteriormente quando testarmos diferentes estratégias de imputação.

Os valores faltantes em PAY_1 são indicados pela string 'Not available'.

9. Crie uma máscara booleana para identificar as linhas que têm esse valor:

```
missing_pay_1_mask = df_clean['PAY_1'] == 'Not available'
```

10. Confirme se o número de amostras com dados faltantes é 3.021 como observado no *Capítulo 1, Exploração e limpeza de dados*:

```
sum(missing_pay_1_mask)
```

A saída deve ser:

3021

Figura 6.7 – Número de linhas com dados faltantes para PAY_1.

O número de linhas com dados faltantes é o esperado.

11. Faça uma cópia dessas linhas em um novo DataFrame que adicionaremos aos dados que usamos na modelagem:

```
df_missing_pay_1 = df_clean.loc[missing_pay_1_mask,:].copy()
```

12. Carregue os dados limpos com os quais trabalhamos nos capítulos 2 a 5. Já removemos as 3.021 amostras com valor faltante para PAY_1 desses dados, logo, é um conjunto de amostras totalmente diferente do que armazenamos em df_missing_pay_1:

Nota: O caminho de seu arquivo de dados limpos pode ser diferente.

```
df = pd.read_csv('../Data/Chapter_1_cleaned_data.csv')
```

13. Crie uma lista com os nomes das colunas desse DataFrame e remova as strings que não fizeram parte do conjunto de características e variável de resposta. Aqui estão os nomes das colunas:

```
df.columns
```

A saída deve ser a seguinte:

```
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_1',
      'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
      'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
      'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
      'default payment next month', 'EDUCATION_CAT', 'graduate school',
      'high school', 'none', 'others', 'university'],
      dtype='object')
```

Figura 6.8 – Nomes das colunas do DataFrame limpo.

14. Atribua os nomes das colunas a uma lista:

```
features_response = df.columns.tolist()
```

15. Crie uma lista de nomes de coluna que não sejam das características ou da reposta para podermos removê-los:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',
                   'EDUCATION_CAT', 'graduate school', 'high school', 'none',
                   'others', 'university']
```

16. Use uma list comprehension para remover os nomes de colunas indesejados e exibir o resultado:

```
features_response = [item for item in features_response if item not in items_to_remove]
features_response
```

A saída será:

```
['LIMIT_BAL',
 'EDUCATION',
 'MARRIAGE',
 'AGE',
 'PAY_1',
 'BILL_AMT1',
 'BILL_AMT2',
 'BILL_AMT3',
 'BILL_AMT4',
 'BILL_AMT5',
 'BILL_AMT6',
 'PAY_AMT1',
 'PAY_AMT2',
 'PAY_AMT3',
 'PAY_AMT4',
 'PAY_AMT5',
 'PAY_AMT6',
 'default payment next month']
```

Figura 6.9 – Nomes de características e da resposta.

Essa lista de nomes de coluna será útil quando estivermos selecionando dados dos DataFrames, inclusive valores faltantes e não faltantes de PAY_1, para combiná-los e testar diferentes estratégias de imputação, o que será feito no próximo exercício.

Exercício 23: Imputação de PAY_1 pela moda e aleatória

Neste exemplo, testaremos algumas das estratégias de imputação mais simples disponíveis para PAY_1 e veremos seus efeitos sobre o desempenho da validação cruzada. As primeiras etapas serão

acrescentar as amostras com valores faltantes para PAY_1 ao final do conjunto de teste com o qual estávamos trabalhando, que não tem valores faltantes para PAY_1. Teremos de embaralhá-las ao executar a validação cruzada para que as amostras com valores faltantes de PAY_1 não fiquem todas no mesmo fold, o que criaria uma situação em que os dados de um dos folds seriam diferentes dos outros.

Embora PAY_1 tenha valores numéricos, é uma característica híbrida entre categórica e numérica como discutimos anteriormente. Vamos tratá-la como categórica para os fins da imputação, já que, se quisermos calcular a média dos valores não faltantes de PAY_1, por exemplo, o resultado pode não ser um inteiro, e não seria interpretável dada a definição de PAY_1. Logo, as estratégias de imputação mais simples que temos disponíveis são a moda e a mediana. Uma estratégia um pouco mais complexa é a seleção aleatória das amostras com valores não faltantes para PAY_1. Vamos examiná-las aqui. Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2PtdRyj>.

1. Importe a classe `train_test_split` do módulo `sklearn` para podermos trabalhar com o mesmo conjunto de teste reservado que estávamos usando:

```
from sklearn.model_selection import train_test_split
```

2. Crie uma divisão 80/20 de treinamento/teste usando o mesmo seed aleatório com o qual trabalhamos:

```
X_train, X_test, y_train, y_test = \
train_test_split(df[features_response[:-1]].values, df['default payment next month'].values,
test_size=0.2, random_state=24)
```

3. Examine as dimensões dos conjuntos de treinamento e teste:

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

A saída deve ser a seguinte:

```
(21331, 17)
(5333, 17)
(21331,)
(5333,)
```


Figura 6.10 – Dimensões dos conjuntos de treinamento e teste com valores não faltantes para PAY_1.

4. Crie os valores de imputação para PAY_1. Primeiro confirme em que índice PAY_1 está na lista de nomes de características e resposta:

```
features_response[4]
```

A saída deve ser:

```
'PAY_1'
```

Figura 6.11 – Confirmando o índice de PAY_1.

Precisaremos desse índice para acessar os valores não faltantes de PAY_1 no array de características de treinamento.

5. Encontre os valores de PAY_1 que poderemos usar na imputação. Primeiro verifique qual é a mediana de PAY_1 usando o código a seguir:

```
np.median(X_train[:,4])
```

A saída deve ser:

```
0.0
```

Figura 6.12 – Mediana de valores não faltantes de PAY_1.

Podemos ver que a mediana dos valores não faltantes de PAY_1 é 0. Você deve se lembrar de que vimos em nossa contagem de valores (value_counts) de PAY_1 que a moda de valores não faltantes também é 0.

Já que a medida e a moda são iguais, só temos duas estratégias de imputação para testar neste exercício: a moda/mediana e a seleção aleatória de valores não faltantes.

6. Crie uma lista de valores para testar na imputação. Podemos usar um único valor numérico, 0, para representar a mediana e a moda, assim como a função random.choice do NumPy para extrair uma amostra aleatória de valores não faltantes de PAY_1:

```
np.random.seed(seed = 1)
```

```
fill_values = [0, np.random.choice(X_train[:,4], size=(3021,), replace=True)]
```

Observe que, para obter a amostra aleatória, fornecemos o seed do gerador de números aleatórios a fim de obter resultados consistentes entre as execuções, indicamos o array de onde será extraída a seleção aleatória (os valores não faltantes de PAY_1 nos dados de treinamento: X_train[:,4]), definimos que o tamanho da amostra aleatória será igual ao da coluna de valores faltantes de

PAY_1 em `df_missing_pay_1` (3.021) e determinamos que queremos usar a amostra com substituição.

7. Crie uma lista de nomes para as estratégias de imputação a fim de ajudá-lo a monitorá-las:

```
fill_strategy = ['mode', 'random']
```

8. Examine o segundo elemento da lista `fill_values`, que é o array de seleções aleatórias de `PAY_1`:

```
fill_values[-1]
```

A saída deve ser:

```
array([ 0,  0,  0, ...,  2,  0, -2])
```

Figura 6.13 – Valores de PAY_1 imputados aleatoriamente.

A saída tem a aparência esperada; esses valores são inteiros no intervalo `[-2, 8]` e sabemos que esse é o formato de `PAY_1`. No entanto, seria melhor visualizar um resumo gráfico de todos os valores imputados para compará-los com a distribuição de `PAY_1`. Isso nos permitiria confirmar se as distribuições são iguais, como se espera da imputação aleatória feita dessa forma.

9. Use histogramas para examinar as distribuições da característica `PAY_1` original sem valores faltantes no conjunto de treinamento e os valores selecionados imputados aleatoriamente:

```
fig, axs = plt.subplots(1,2, figsize=(8,3))
bin_edges = np.arange(-2,9)
axs[0].hist(X_train[:,4], bins=bin_edges, align='left')
axs[0].set_xticks(bin_edges)
axs[0].set_title('Non-missing values of PAY_1')
axs[1].hist(fill_values[-1], bins=bin_edges, align='left')
axs[1].set_xticks(bin_edges)
axs[1].set_title('Random selection for imputation')
plt.tight_layout()
```

Esse código deve produzir o gráfico mostrado na Figura 6.14.

Essas duas distribuições parecem muito semelhantes. Só a escala do eixo y é que indica que há menos valores imputados do que os valores que existem no dataset original. Isso mostra que selecionamos valores para `PAY_1` que imitam fielmente a frequência relativa de diferentes valores dessa característica nos dados. Estamos prontos para definir a validação cruzada que usaremos para comparar os métodos de imputação.

Não precisaremos fazer uma busca de hiperparâmetros com

validação cruzada, como fizemos com a função `GridSearchCV` do `scikit-learn`. Já sabemos que a floresta aleatória é o modelo que usaremos e os hiperparâmetros que devemos empregar. Queremos executar a validação cruzada com um único conjunto de hiperparâmetros para ter várias estimativas da pontuação de teste fora da amostra. Para fazê-lo, podemos usar uma classe semelhante, porém mais simples, chamada `cross_validate`, junto com a classe `KFold`. Essas classes podem nos ajudar a executar a validação cruzada como fizemos com `GridSearchCV`, mas não envolvem uma busca na grade de hiperparâmetros.

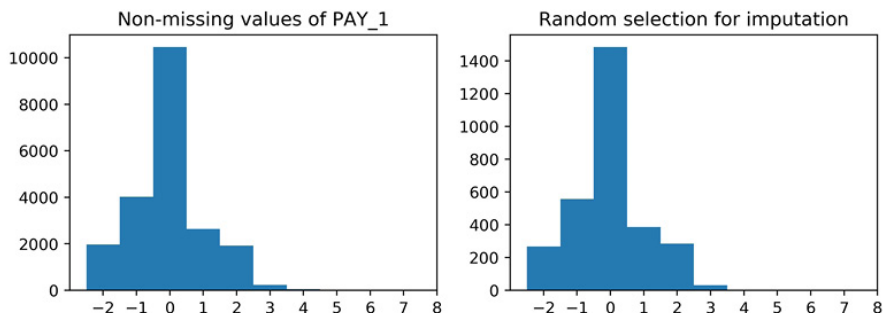


Figura 6.14 – Distribuições de PAY_1 e os valores imputados aleatoriamente.

10. Importe a classe `KFold` com este código:

```
from sklearn.model_selection import KFold
```

11. Instancie a classe `KFold` desta forma:

```
k_folds = KFold(n_splits=4, shuffle=True, random_state=1)
```

Especificamos o uso de quatro folds no conjunto de treinamento, como fizemos anteriormente ao executar a validação cruzada. Também estamos definindo que queremos embaralhar os dados antes de dividi-los nos folds. Isso é importante porque acrescentaremos as amostras com valores imputados ao fim dos arrays de treinamento de características e resposta. No entanto, quando executarmos a validação cruzada, queremos que essas amostras sejam “misturadas” em todos os quatro folds para que cada fold tenha alguns valores imputados. Também definimos o seed aleatório de repetição.

12. Importe a classe `cross_validate`:

```
from sklearn.model_selection import cross_validate
```

13. Importe a classe do classificador de floresta aleatória:

from sklearn.ensemble import RandomForestClassifier

14. Instancie a classe de floresta aleatória, usando os hiperparâmetros que determinamos que são melhores para os dados do estudo de caso: 200 árvores com profundidade máxima igual a 9:

```
rf = RandomForestClassifier\  
(n_estimators=200, criterion='gini', max_depth=9,  
min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0,  
max_features='auto', max_leaf_nodes=None, min_impurity_decrease=0.0,  
min_impurity_split=None, bootstrap=True, oob_score=False, n_jobs=None,  
random_state=4, verbose=1, warm_start=False, class_weight=None)
```

Embora estejamos trabalhando com dados um pouco diferentes, devido aos valores imputados, assumiremos que os hiperparâmetros que determinamos para a floresta aleatória continuam sendo os melhores. Agora passaremos para a validação cruzada. É uma célula de código longa, portanto, a dividiremos em várias etapas.

15. Abra um loop for. Esse loop operará com as diferentes estratégias de imputação que temos, que são duas:

```
for counter in range(len(fill_values)):
```

16. Dentro do loop for, a primeira etapa é criar uma cópia do DataFrame de amostras com os valores faltantes de PAY_1 e, em seguida, preencher esses valores com a estratégia de imputação que estivermos considerando:

```
#Copia o dataframe de valores faltantes de PAY_1 e atribui os valores imputados  
df_fill_pay_1_filled = df_missing_pay_1.copy()  
df_fill_pay_1_filled['PAY_1'] = fill_values[counter]
```

17. Divida os dados imputados nos conjuntos de treinamento e teste. Aqui, só usaremos a parte de treinamento na validação cruzada. Essa operação funciona da mesma forma de quando dividimos os dados limpos com os valores faltantes removidos:

```
#Divide os dados imputados em treinamento e teste, usando a mesma  
#divisão 80/20 que empregamos com os valores não faltantes de PAY_1  
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_test = \  
train_test_split(  
    df_fill_pay_1_filled[features_response[:-1]].values,  
    df_fill_pay_1_filled['default payment next month'].values,  
    test_size=0.2, random_state=24)
```

18. Agora queremos combinar os dados imputados com os dados não faltantes de PAY_1 com os quais trabalhamos. Vamos concatená-los:

```
#Concatena os dados imputados com o array de dados não faltantes  
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)  
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)
```

19. Para concluir, juntaremos tudo. Usaremos esses dados com o procedimento `cross_validation`, assim como com o divisor `KFolds` e um modelo de floresta aleatória. Exiba a pontuação média do teste, com o desvio padrão, para cada método.

```
#Usa o divisor KFolds e o modelo de floresta aleatória para obter
#a pontuação da validação cruzada de 4 folds para os dois métodos de imputação
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all,
    scoring='roc_auc', cv=k_folds, n_jobs=-1, verbose=1,
    return_train_score=True, return_estimator=True,
    error_score='raise-deprecating')

test_score = imputation_compare_cv['test_score']
print(fill_strategy[counter] + 'imputation: ' +
    'mean testing score ' + str(np.mean(test_score)) +
    ', std ' + str(np.std(test_score)))
```

Terminada a execução, a saída do loop for deve ser semelhante a esta:

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done 2 out of 4 | elapsed: 11.7s remaining: 11.7s
[Parallel(n_jobs=-1)]: Done 4 out of 4 | elapsed: 11.8s finished
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.

mode imputation: mean testing score 0.772866246168149, std 0.0031479941297533737
random imputation: mean testing score 0.7692540439833129, std 0.003660875187678248

[Parallel(n_jobs=-1)]: Done 2 out of 4 | elapsed: 19.9s remaining: 19.9s
[Parallel(n_jobs=-1)]: Done 4 out of 4 | elapsed: 19.9s finished
```

Figura 6.15 – Saída do loop de validação cruzada que testa diferentes estratégias de imputação.

Essa é a etapa em que o ajuste do modelo e o teste de validação cruzada são conduzidos em segundo plano e ocultados da visualização pela classe `cross_validation`. Observe que fornecemos muitas informações para essa classe: o modelo que está passando pela validação cruzada (`rf`), os dados `X_train_all` e `y_train_all`, e o divisor `KFolds`, entre outras opções. Uma delas é `n_jobs=-1`, indicando a execução de tarefas em paralelo para que tudo ocorra mais rapidamente.

A saída exibida aqui indica que essas duas estratégias de imputação têm desempenhos semelhantes. No entanto, a estratégia de imputação mais simples, apenas preencher todos os valores faltantes de `PAY_1` com o valor mais comum das amostras não faltantes, que é 0, tem melhor desempenho: a pontuação média de teste (ROC AUC) entre os quatro folds é 0.773 para a imputação da moda, versus 0.769 com o uso da imputação aleatória.

E quanto à pontuação média de teste obtida aqui, na qual incluímos as

amostras com os dados faltantes, versus a pontuação média de teste que observamos quando usamos apenas os dados em que todos os valores das características eram conhecidos? Ela foi de 0.776. Logo, vimos uma pequena degradação na estimativa do desempenho de teste do modelo fora da amostra. Isso era esperado, já que agora estamos incluindo dados que têm problemas e incertezas. Porém, ainda está dentro das 0.006 unidades de ROAC AUC da melhor pontuação que observamos ao trabalhar com os dados não faltantes, então, não é tão baixa. Devemos estar confiantes de que encontramos uma maneira aceitável de fazer previsões para contas que têm valores faltantes para PAY_1, como o cliente solicitou. É possível aumentar ainda mais a qualidade da previsão com dados imputados? Veremos se conseguimos fazer isso na próxima seção.

Um modelo preditivo para PAY_1

A maneira mais precisa, porém também a mais trabalhosa, de imputar uma característica com valores faltantes é criar um modelo preditivo para ela. Você pode considerar isso como uma extensão natural dos métodos mais simples, usando cada vez mais informações do dataset ao imputar os valores faltantes:

- Os métodos mais simples como a média, a mediana, a moda e a seleção aleatória usam apenas informações de valores não faltantes da característica.
- Métodos intermediariamente complexos como a interpolação usam informações adicionais sobre quanto uma amostra específica está “próxima” de outras amostras, no contexto espacial ou temporal, e beneficiam-se disso para fornecer uma imputação mais precisa.
- Para concluir, a imputação baseada em modelo usa todas as características sem os valores faltantes, para prever os valores faltantes da característica afetada. Logo, ela é semelhante a qualquer outro modelo preditivo, onde consideramos a característica com valores faltantes como a variável de resposta.

Ao considerar como criar um modelo preditivo para PAY_1, pondere em que esse modelo seria semelhante a, ou diferente de, outros modelos que usamos.

Os diferentes tipos de modelos supervisionados de machine learning são a regressão e a classificação. Como discutimos anteriormente, não consideramos PAY_1 como uma característica numérica. Calcular uma média para a imputação, que pode levar a um valor decimal, não faz

sentido de acordo com a definição dessa característica. Encontraremos esse mesmo problema se tentarmos modelar PAY_1 em uma escala contínua, como seria feito com o uso de algoritmos de regressão. Logo, ao considerar que tipo de modelo de aprendizado supervisionado devemos usar para PAY_1, sabemos que ele deve ser um modelo de classificação.

No entanto, o modelo de classificação de PAY_1 será diferente do modelo de classificação da inadimplência de contas de crédito que construímos para os dados do estudo de caso. Isso ocorre porque, ao contrário do problema de não pagamento de dívidas, em que a variável de resposta só pode assumir dois valores (a conta está ou não está inadimplente), há mais de dois valores para PAY_1. Eles incluem todos os níveis dessa variável, de -2 a 8. Então, em vez de um problema de classificação binária, o modelo de PAY_1 será um problema de classificação multiclasse.

Percorreremos as etapas de definição e treinamento desse problema de classificação multiclasse. Exceto por algumas diferenças importantes, você verá que o processo é muito semelhante à criação de um modelo de classificação binária.

Primeiro, crie uma cópia do DataFrame limpo com os valores não faltantes de PAY_1.

```
pay_1_df = df.copy()
```

Isso representa os dados disponíveis para treinamento e teste, já que sabemos o valor da variável de resposta, PAY_1 nesse caso, para todas essas amostras.

Semelhante a como fizemos em nossa abordagem anterior para a seleção de colunas, criaremos uma lista de nomes de colunas que serão características.

```
features_for_imputation = pay_1_df.columns.tolist()
```

Criaremos um subconjunto dessa lista para resumi-la à lista de características do modelo de imputação removendo os nomes de todos os metadados, assim como 'default payment next month', de que não precisaremos aqui, e 'PAY_1', que é a variável de resposta desse problema:

```
items_to_remove = ['ID', 'SEX', 'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6',  
                   'EDUCATION_CAT', 'graduate school', 'high school', 'none',  
                   'others', 'university', 'default payment next month', 'PAY_1']
```

Remova esses itens usando uma list comprehension:

```
features_for_imputation = [item for item in features_for_imputation if item not in
    items_to_remove]
features_for_imputation
```

A saída deve conter apenas as características exceto PAY_1:

```
[ 'LIMIT_BAL',
  'EDUCATION',
  'MARRIAGE',
  'AGE',
  'BILL_AMT1',
  'BILL_AMT2',
  'BILL_AMT3',
  'BILL_AMT4',
  'BILL_AMT5',
  'BILL_AMT6',
  'PAY_AMT1',
  'PAY_AMT2',
  'PAY_AMT3',
  'PAY_AMT4',
  'PAY_AMT5',
  'PAY_AMT6' ]
```

Figura 6.16 – Características do modelo preditivo de PAY_1.

Agora que preparamos os nomes das variáveis para o modelo de imputação, percorreremos as etapas restantes para a construção desse modelo como um exercício.

Exercício 24: Construindo um modelo de classificação multiclasse para a imputação

Neste exercício, construiremos um modelo multiclasse para a imputação. Ao construí-lo, assumiremos que a floresta aleatória é o melhor modelo a ser usado, em parte devido às nossas experiências com esse dataset, e também porque o algoritmo de floresta aleatória do scikit-learn suporta imediatamente a classificação multiclasse. Há várias maneiras de usar a regressão logística e outros algoritmos de classificação binária para a classificação multiclasse. No entanto, não entraremos nesses detalhes aqui.

Nota: Duas abordagens que você pode ficar interessado em conhecer, se quiser ganhar mais experiência na classificação multiclasse, são a um contra todos e a um contra um (<https://scikit-learn.org/stable/modules/multiclass.html>). Essas abordagens permitem que modelos de classificação binária, como a regressão logística, sejam usados para a classificação multiclasse.

Seguiremos as etapas habituais de criar uma divisão de treinamento/teste, executar uma busca de parâmetros com a validação cruzada usando os dados de treinamento e confirmar o desempenho do modelo

no conjunto de teste. Já conhecemos a maioria delas, no entanto há a peculiaridade adicional de estarmos modelando um problema multiclasse. Selecionaremos a acurácia como métrica de desempenho do modelo, já que ela é um pouco mais fácil de usar com problemas multiclasse. Contudo, há maneiras de estender métricas baseadas em limite como a ROC AUC para configurações multiclasse.

Nota: Para obter mais informações sobre o uso da ROC AUC na classificação multiclasse, acesse https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html.

Execute as etapas a seguir para fazer o exercício:

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2PtdRyj>.

1. Crie uma divisão de treinamento e teste nos dados do modelo de imputação:

```
X_impute_train, X_impute_test, y_impute_train, y_impute_test = \
train_test_split(
    pay_1_df[features_for_imputation].values,
    pay_1_df['PAY_1'].values,
    test_size=0.2, random_state=24)
```

Aqui, PAY_1 é a variável de resposta.

2. Selecione uma grade de hiperparâmetros para a pesquisa com a validação cruzada:

```
rf_impute_params = {'max_depth':[3, 6, 9, 12],
                    'n_estimators':[10, 50, 100, 200]}
```

Esses parâmetros são os mesmos usados para o modelo de não pagamento de dívidas.

3. Importe e instancie a classe GridSearchCV:

```
from sklearn.model_selection import GridSearchCV
cv_rf_impute = GridSearchCV(rf, param_grid=rf_impute_params, scoring='accuracy',
fit_params=None, n_jobs=-1, iid=False, refit=True,
cv=4, verbose=2, error_score=np.nan, return_train_score=True)
```

Observe que estamos reutilizando a instância de modelo de floresta aleatória rf do modelo de contas de crédito inadimplentes. Todas as opções devem usar seus padrões exceto n_estimators e max_depth, que serão alteradas pelo processo de busca em grade.

Ao instanciar a classe GridSearchCV, agimos como antes, mas aqui usamos o método de acurácia como mencionado anteriormente.

Também estamos permitindo o processamento paralelo com `n_jobs = -1`.

4. Execute a busca em grade com este código:

```
cv_rf_impute.fit(X_impute_train, y_impute_train)
```

A saída deve ser:

```
Fitting 4 folds for each of 16 candidates, totalling 64 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.  
[Parallel(n_jobs=-1)]: Done 33 tasks | elapsed: 21.8s  
[Parallel(n_jobs=-1)]: Done 64 out of 64 | elapsed: 1.0min finished  
[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.  
[Parallel(n_jobs=1)]: Done 100 out of 100 | elapsed: 5.1s finished
```

```
GridSearchCV(cv=4, error_score=nan,  
             estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
                                               max_depth=9, max_features='auto', max_leaf_nodes=None,  
                                               min_impurity_decrease=0.0, min_impurity_split=None,  
                                               min_samples_leaf=1, min_samples_split=2,  
                                               min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,  
                                               oob_score=False, random_state=4, verbose=1, warm_start=False),  
             fit_params=None, iid=False, n_jobs=-1,  
             param_grid={'max_depth': [3, 6, 9, 12], 'n_estimators': [10, 50, 100, 200]},  
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,  
             scoring='accuracy', verbose=2)
```

Figura 6.17 – Saída da busca em grade com validação cruzada para o modelo de imputação.

Repare que não tivemos de fazer nada de especial aqui por estarmos ajustando um modelo de classificação multiclasse. A implementação do scikit-learn para a floresta aleatória manipula automaticamente os dados multiclasse sem que seja necessária nenhuma preparação.

5. Observe os hiperparâmetros do melhor modelo de validação cruzada:

```
cv_rf_impute.best_params_
```

A saída deve ser:

```
{'max_depth': 12, 'n_estimators': 100}
```

Figura 6.18 – Hiperparâmetros ótimos da validação cruzada do modelo de imputação.

6. Veja qual é a pontuação de acurácia do melhor modelo:

```
cv_rf_impute.best_score_
```

A saída será:

```
0.7337676389523727
```

Figura 6.19 – Melhor pontuação média de acurácia da validação cruzada.

Como podemos interpretar essa pontuação de acurácia? Considere o método de imputação que usa a moda de valores não faltantes de

PAY_1, que é 0, para todos os valores faltantes. Qual seria a acurácia da imputação pela moda?

7. Examine as contagens de valores de PAY_1 em pay_1_df para ver a frequência relativa dos diferentes valores:

```
pay_1_value_counts = pay_1_df['PAY_1'].value_counts().sort_index()
pay_1_value_counts
```

A saída deve ser:

-2	2476
-1	5047
0	13087
1	3261
2	2378
3	292
4	63
5	23
6	11
7	9
8	17

Figura 6.20 – Contagens de valores de PAY_1.

8. Exiba a frequência relativa dos valores de PAY_1 como frações da seguinte forma:

```
pay_1_value_counts/pay_1_value_counts.sum()
```

A saída será:

-2	0.092859
-1	0.189281
0	0.490812
1	0.122300
2	0.089184
3	0.010951
4	0.002363
5	0.000863
6	0.000413
7	0.000338
8	0.000638

Figura 6.21 – Frações de valores de PAY_1.

Podemos ver que a moda é, obviamente, o valor predominante em PAY_1. Também podemos ver pelas frações da *Figura 6.21* que 49% dos valores são iguais à moda. Logo, usando a imputação pela moda, o esperado é que o valor da imputação esteja correto aproximadamente 49% do tempo. No entanto, usando um modelo para imputar os valores de PAY_1, nossos resultados na etapa 6 indicam que podemos estar corretos 73% do tempo. Então, *um benefício do uso da imputação baseada em modelo é que podemos criar valores de preenchimento mais precisos para os valores desconhecidos de uma característica.*

9. Para verificar se a acurácia da validação cruzada pode ser generalizada para o conjunto de teste, temos de fazer previsões com ele:

```
y_impute_predict = cv_rf_impute.predict(X_impute_test)
```

10. Importe a classe `accuracy_score` e veja a acurácia das previsões de teste:

```
from sklearn import metrics  
metrics.accuracy_score(y_impute_test, y_impute_predict)
```

A saída deve ser:

0.7387961747609225

Figura 6.22 – Acurácia do modelo de imputação em um conjunto reservado para teste.

A acurácia de 74% no conjunto de teste é comparável a, e um pouco mais alta que, a acurácia média de 73% da validação cruzada.

Seria bom verificar a distribuição de classes previstas desse modelo, semelhante a como fizemos para o método de imputação aleatória. Sabemos que o modelo tem cerca de 73% de acurácia, mas é sempre bom visualizar suas previsões.

11. Use este código para plotar os valores e as previsões reais do conjunto reservado para teste:

```
fig, axs = plt.subplots(1,2, figsize=(8,3))  
axs[0].hist(y_impute_test, bins=bin_edges, align='left')  
axs[0].set_xticks(bin_edges)  
axs[0].set_title('Non-missing values of PAY_1')  
axs[1].hist(y_impute_predict, bins=bin_edges, align='left')  
axs[1].set_xticks(bin_edges)  
axs[1].set_title('Model-based imputation')  
plt.tight_layout()
```

O gráfico resultante pode ser visualizado na Figura 6.23.

A mostra que o modelo de imputação tem mais probabilidade de prever um valor 0 para PAY_1 do que deveria. No entanto, ele prevê outros valores e consequentemente é mais preciso que a imputação pela moda.

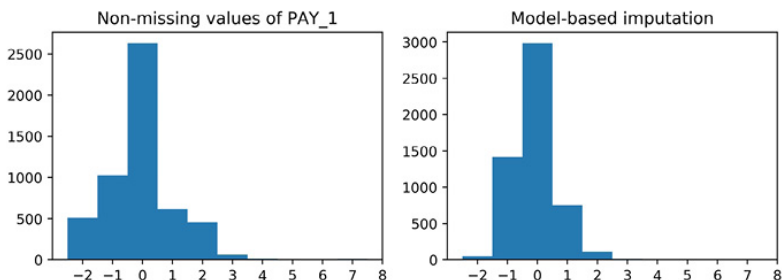


Figura 6.23 – Valores reais e previstos de PAY_1 para o conjunto reservado para teste.

Nosso modelo de imputação foi testado e os hiperparâmetros selecionados estão confirmados. Após o conjunto reservado para teste ter sido usado na verificação do desempenho do modelo com dados desconhecidos, os dados de treinamento e teste devem ser reunidos para treinarmos um modelo com o máximo de dados possível. Normalmente o aumento dos dados de treinamento resulta em um modelo melhor.

12. Colete todos os valores com PAY_1 conhecido para treinar a versão final do modelo de imputação:

```
X_impute_all = pay_1_df[features_for_imputation].values  
y_impute_all = pay_1_df['PAY_1'].values
```

13. Defina um modelo de floresta aleatória com os hiperparâmetros ótimos para imputação:

```
rf_impute = RandomForestClassifier(n_estimators=100, max_depth=12)
```

14. Ajuste o modelo de imputação com todos os dados disponíveis:

```
rf_impute.fit(X_impute_all, y_impute_all)
```

A saída deve ser:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
                        max_depth=12, max_features='auto', max_leaf_nodes=None,  
                        min_impurity_decrease=0.0, min_impurity_split=None,  
                        min_samples_leaf=1, min_samples_split=2,  
                        min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,  
                        oob_score=False, random_state=None, verbose=0,  
                        warm_start=False)
```

Figura 6.24 – Saída do treinamento do modelo de imputação.

Esse modelo está pronto para ser usado para a imputação.

Usando o modelo de imputação e comparando-o com outros métodos

Criamos o modelo para a imputação baseada em modelo, portanto, podemos usar valores imputados por modelo para PAY_1 na validação cruzada com o modelo de inadimplência de contas de crédito e ver qual é o desempenho em comparação com os métodos de imputação mais simples que já testamos. Para começar, faça uma cópia do DataFrame com os valores faltantes de PAY_1:

```
df_fill_pay_1_model = df_missing_pay_1.copy()
```

Examine os valores de PAY_1 para confirmar se eles representam os

dados faltantes:

```
df_fill_pay_1_model['PAY_1'].head()
```

A saída deve ser:

```
17    Not available
28    Not available
29    Not available
54    Not available
60    Not available
Name: PAY_1, dtype: object
```

Figura 6.25 – Dados faltantes de PAY_1.

Agora, substitua os valores faltantes pelas previsões do modelo de imputação:

```
df_fill_pay_1_model['PAY_1'] =
    rf_impute.predict(df_fill_pay_1_model[features_for_imputation].values)
```

Examine os valores de PAY_1 para confirmar se eles foram imputados:

```
df_fill_pay_1_model['PAY_1'].head()
```

A saída será:

```
17    0
28   -1
29    0
54    0
60    0
Name: PAY_1, dtype: int64
```

Figura 6.26 – Valores de PAY_1 imputados por modelo.

Isso mostra que os valores foram imputados como desejado. É possível ver pela saída da Figura 6.26 que nem todos os valores são iguais, como esperávamos. Podemos fazer uma verificação rápida das previsões usando o método `value_counts`:

```
df_fill_pay_1_model['PAY_1'].value_counts().sort_index()
```

Esta é a saída:

```
-2      30
-1     763
0     1715
1      438
2       64
3        7
4         2
6         1
8         1
```

Figura 6.27 – Previsões do modelo de imputação.

Na Figura 6.27, podemos ver que há um intervalo de valores de

previsão de PAY_1 fornecidos pelo modelo de imputação que estão aproximadamente dentro da frequência relativa esperada. Temos de pegar uma amostra de 80% deles para combinar com os dados de treinamento do modelo do estudo de caso e examinar como esse método de imputação afeta o desempenho do modelo, como fizemos para a imputação pela moda e aleatória.

Divida os dados imputados por modelo da seguinte forma:

```
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_test = \
train_test_split(
    df_fill_pay_1_model[features_response[:-1]].values,
    df_fill_pay_1_model['default payment next month'].values,
    test_size=0.2, random_state=24)
```

Combine-os com os dados conhecidos de PAY_1 como antes:

```
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)
```

Confirme se o objeto rf ainda contém o modelo com hiperparâmetros ótimos para o problema do estudo de caso:

```
rf
```

A saída é a seguinte:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
    max_depth=9, max_features='auto', max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,
    oob_score=False, random_state=4, verbose=1, warm_start=False)
```

Figura 6.28 – Confirmando o modelo para o estudo de caso.

O objeto rf contém um classificador de floresta aleatória com 200 árvores e profundidade máxima igual a 9, como esperado. Use esse modelo com a combinação de dados de treinamento imputados e não faltantes na validação cruzada como fizemos com os outros métodos de imputação para testar a imputação baseada em modelo:

```
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all, scoring='roc_auc',
cv=k_folds, n_jobs=-1, verbose=1,
return_train_score=True, return_estimator=True,
error_score='raise-deprecating')
```

Quando esse processo terminar, você poderá examinar a pontuação média de teste do modelo entre os quatro folds:

```
np.mean(imputation_compare_cv['test_score'])
```

A saída deve ser:

Figura 6.29 – Pontuação média de teste na validação cruzada com o uso de valores de PAY_1 imputados por modelo.

O que podemos dizer do resultado da imputação baseada em modelo em comparação com os métodos mais simples pela moda e aleatório? A ROC AUC média da validação cruzada igual a 0.7727 chega muito perto de, mas é um pouco pior que, nosso melhor resultado anterior de 0.7729 obtido na imputação baseada na moda.

Tanto a imputação baseada na moda quanto a baseada em modelo têm basicamente o mesmo desempenho. Observe que o desvio padrão da pontuação da validação cruzada com a imputação baseada na moda calculado acima é de 0.003, logo, a pontuação média da imputação baseada em modelo está dentro do desvio padrão igual a um do melhor método. Qual devemos usar?

Do ponto de vista da simplicidade, a imputação baseada na moda é claramente melhor. O código que a implementa é muito mais conciso do que o da imputação baseada em modelo que acabamos de concluir. A única razão para usarmos a imputação baseada em modelo seria se o cliente soubesse que também podemos fazer previsões para PAY_1 e ele as quisesse. Consultamos nosso cliente a esse respeito e soubemos que ele só está interessado nas previsões de não pagamento de dívidas. Logo, daremos prosseguimento com o método de imputação mais simples.

Primeiro, reatribua os valores preenchidos usando a imputação com a moda igual a 0:

```
df_fill_pay_1_model['PAY_1'] = np.zeros_like(df_fill_pay_1_model['PAY_1'].values)
```

A função `zeros_like` do Numpy cria um array de zeros com a mesma dimensão do array de entradas. Verifique se funcionou:

```
df_fill_pay_1_model['PAY_1'].unique()
```

A saída deve ser:

```
array([0])
```

Figura 6.30 – Confirme a imputação pela moda.

Deve haver apenas valores 0 na imputação baseada na moda, e podemos ver na *Figura 6.30* que funcionou, já que há um único valor exclusivo para PAY_1 e ele é 0. Agora repetiremos a divisão dos dados imputados:

```
X_fill_pay_1_train, X_fill_pay_1_test, y_fill_pay_1_train, y_fill_pay_1_test = \
```



```
train_test_split(  
    df_fill_pay_1_model[features_response[:-1]].values,  
    df_fill_pay_1_model['default payment next month'].values,  
    test_size=0.2, random_state=24)
```

e os combinaremos com os dados não faltantes, dessa vez incluindo o conjunto de teste desconhecido que usaremos em breve:

```
X_train_all = np.concatenate((X_train, X_fill_pay_1_train), axis=0)  
X_test_all = np.concatenate((X_test, X_fill_pay_1_test), axis=0)  
y_train_all = np.concatenate((y_train, y_fill_pay_1_train), axis=0)  
y_test_all = np.concatenate((y_test, y_fill_pay_1_test), axis=0)
```

Apenas para confirmar se os dados foram restaurados para o estado em que se encontravam quando testamos a imputação baseada na moda anteriormente, executaremos a validação cruzada e veremos se obtivemos a mesma pontuação:

```
imputation_compare_cv = cross_validate(rf, X_train_all, y_train_all, scoring='roc_auc',  
cv=k_folds, n_jobs=-1, verbose=1,  
return_train_score=True, return_estimator=True,  
error_score='raise-deprecating')
```

Após executar a validação cruzada, calcule a pontuação média:

```
np.mean(imputation_compare_cv['test_score'])
```

O resultado deve ser:

0.772866246168149

Figura 6.31 – Confirmando a pontuação da validação cruzada com a imputação pela moda.

Concluimos a seleção da estratégia de imputação.

Confirmando o desempenho do modelo com o conjunto de teste desconhecido

Quase terminamos de construir o modelo para os dados do estudo de caso. A última etapa do processo de construção é examinar seu desempenho com o conjunto de teste, que não usamos em nenhuma atividade de construção de modelos até agora.

Você deve ter notado que na verdade já usamos as amostras do conjunto de teste em alguns locais. Ao executar a exploração e a limpeza iniciais dos dados, ainda não as tínhamos separado. Isso é comum, já que queremos nos certificar de que os dados de teste estejam limpos da mesma forma que os dados de treinamento. Outro momento em que usamos as amostras de teste foi agora, quando

fizemos previsões para PAY_1 e as comparamos com valores reais. No entanto, consideramos isso como um esforço de modelagem separado.

A principal motivação existente por trás do uso de um conjunto de teste “desconhecido” é termos uma fonte de dados independente com a qual testar o modelo que selecionamos com os hiperparâmetros que escolhemos para ele. Usamos todos os dados de treinamento para esse fim. Embora possamos dizer que estimamos o desempenho fora da amostra usando a validação cruzada, o conjunto de teste fornece uma maneira de vermos realmente como o modelo se sai com dados que nunca foram usados para ajustá-lo. Idealmente, os resultados serão semelhantes às estimativas da validação cruzada. Se não forem, temos de saber o porquê: os dados de teste são de alguma forma diferentes dos dados de treinamento? Visualizações das características e da resposta seriam úteis nessa pesquisa.

Agora que combinamos todos os dados não imputados e imputados para treinamento e teste, podemos treinar o modelo final desta forma:

```
rf.fit(X_train_all, y_train_all)
```

A saída deve ser:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
    max_depth=9, max_features='auto', max_leaf_nodes=None,  
    min_impurity_decrease=0.0, min_impurity_split=None,  
    min_samples_leaf=1, min_samples_split=2,  
    min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=None,  
    oob_score=False, random_state=4, verbose=1, warm_start=False)
```

Figura 6.32 – Treinando o modelo final.

Para concluir, faremos previsões de não pagamento de dívidas com o conjunto de teste. Temos de prever probabilidades para poder calcular a ROC AUC:

```
y_test_all_predict_proba = rf.predict_proba(X_test_all)
```

Importe a função `roc_auc_score` e calcule a pontuação de teste:

```
from sklearn.metrics import roc_auc_score  
roc_auc_score(y_test_all, y_test_all_predict_proba[:,1])
```

A saída é:

```
0.7696243835824927
```

Figura 6.33 – Pontuação de teste do modelo final com dados imputados e não imputados.

O que a pontuação de teste revelou? A comparação apropriada a ser feita é com a pontuação de teste da validação cruzada que calculamos

usando valores de `PAY_1` imputados pela moda, que foi de 0.773. Podemos ver que a pontuação de teste aqui, de 0.770, está aproximadamente dentro do desvio padrão igual a 1 da estimativa da validação cruzada. Concluimos que o modelo é robusto e está pronto para distribuição para o cliente.

As atividades de construção do modelo terminaram. Uma última etapa antes da distribuição de um modelo treinado do `scikit-learn` seria ajustá-lo com todos os dados disponíveis, inclusive o conjunto de teste desconhecido. Isso seria feito com a concatenação das características (`X_train_all`, `X_test_all`) e rótulos (`y_train_all`, `y_test_all`) dos dados de treinamento e teste e seu uso para ajustar o modelo.

Nossas principais preocupações agora são ajudar o cliente a usar o modelo para atingir suas metas empresariais, fornecer orientação sobre como o desempenho do modelo pode ser monitorado com o passar do tempo e apresentar todos os nossos resultados e recomendações para o cliente.

Análise financeira

As métricas de desempenho de modelos que calculamos até agora foram baseadas em medidas abstratas que poderiam ser aplicadas na análise de qualquer modelo de classificação: qual é a acurácia de um modelo, qual sua capacidade para identificar verdadeiros positivos em relação aos falsos positivos (ROC AUC) ou qual a exatidão das previsões positivas (precisão). Essas métricas são importantes para conhecermos o funcionamento básico de um modelo e são amplamente usadas na comunidade de machine learning, logo, é necessário entendê-las bem. Contudo, para a aplicação de um modelo a processos empresariais, nem sempre os clientes conseguem empregar as métricas de desempenho para estabelecer como usarão o modelo como guia para a tomada de decisões operacionais ou saber quanto valor ele pode gerar. Para darmos um passo além e fazermos a conexão do universo matemático das probabilidades previstas e limites com o universo empresarial de custos e benefícios, geralmente uma análise financeira de algum tipo é requerida.

Para ajudar o cliente nessa análise, o cientista de dados precisa saber que tipos de decisões e medidas devem ser tomados conforme as previsões feitas pelo modelo. Esse deve ser o tópico de uma conversa com o cliente, preferivelmente no início do ciclo de vida do projeto. Deixamos isso para o fim do livro a fim de podermos estabelecer uma

compreensão básica do que é a modelagem preditiva e como ela funciona. No entanto, *conhecer o contexto empresarial em que será usado o modelo no começo de um projeto permite definir objetivos para o desempenho em relação ao valor que se pretende gerar durante o desenvolvimento do modelo*. Converter métricas de desempenho do modelo para o linguajar financeiro é o tópico desta seção.

Para um modelo de classificação binária como o do estudo de caso, aqui estão algumas perguntas para as quais os cientistas de dados precisam conhecer as respostas a fim de descobrir como aplicar o modelo para o cliente:

- O cliente deseja usar o modelo para ajudá-lo a tomar que tipos de decisões?
- Como as probabilidades previstas de um modelo de classificação binária podem ser usadas para ajudar a tomar essas decisões?
- São decisões de resposta sim/não? Se forem, a seleção de um único limite de probabilidade prevista será suficiente.
- Há mais de dois níveis de atividade envolvidos na tomada de decisão de acordo com os resultados do modelo? Então, selecionar 2 ou mais limites, por exemplo, para classificar as previsões em de baixo, médio ou alto risco, pode ser a solução. Nesse caso, as probabilidades previstas abaixo de 0.5 poderiam ser consideradas de baixo risco, as entre 0.5 e 0.75 teriam médio risco e as acima de 0.75 seriam de alto risco.
- Quais são os custos da adoção de todos os diferentes cursos de ação que estão disponíveis, com base na orientação do modelo?
- Quais são os possíveis benefícios obtidos com as ações bem-sucedidas executadas como resultado da orientação do modelo?

Conversa financeira com o cliente

Perguntamos ao cliente do estudo de caso sobre os pontos descritos acima e soubemos o seguinte: para contas de crédito que estejam em alto risco de inadimplência, o cliente está projetando um novo programa para fornecer aconselhamento individualizado para o titular da conta de modo a encorajá-lo a pagar sua dívida a tempo ou oferecer opções de pagamento se isso não for possível. O aconselhamento sobre crédito é feito por representantes treinados do serviço de atendimento ao cliente que trabalham em um call center. O custo por sessão de aconselhamento é de NT\$7,500 e a taxa de sucesso

esperada de uma sessão é a de que 70% dos recebedores de chamadas telefônicas oferecendo aconselhamento paguem sua dívida a tempo ou façam acordos alternativos aceitos pelo credor. Os possíveis benefícios do aconselhamento bem-sucedido são que o valor da cobrança mensal de uma conta será percebido como economia, se ela fosse ficar inadimplente, mas não ficou como resultado da conversa. Atualmente, as cobranças mensais de contas inadimplentes são relatadas como perdas.

Após ter a conversa acima com o cliente, estaremos de posse dos materiais de que precisamos para fazer uma análise financeira. O cliente deseja que o ajudemos a decidir que membros contatar e oferecer aconselhamento de crédito. Se pudermos ajudá-lo a reduzir a lista de pessoas que serão contatadas para aconselhamento, estaremos ajudando-o a economizar evitando contatos desnecessários e caros. Os recursos limitados do cliente para o aconselhamento serão gastos mais apropriadamente em contas que tiverem risco mais alto de inadimplir. Isso deve gerar uma economia maior por evitar a inadimplência. O cliente também nos informou que nossa análise pode ajudá-lo a solicitar um orçamento para o programa de aconselhamento, se pudermos lhe dar uma ideia de quantas sessões valeria a pena oferecer.

Ao passar para a análise financeira, vemos que a decisão que o modelo ajudará o cliente a tomar para cada conta é do tipo sim/não: ele deve oferecer aconselhamento para o titular de uma conta específica? Logo, nossa análise deve dar ênfase a encontrar um limite de probabilidade prevista apropriado segundo o qual possamos dividir nossas contas em dois grupos: contas de maior risco que receberão aconselhamento e as de menor risco que não receberão.

Exercício 25: Caracterizando custos e economias

A conexão entre a saída do modelo e as decisões empresariais que o cliente tomará se resume à seleção de um limite para a probabilidade prevista do modelo. Portanto, neste exercício, caracterizaremos os custos esperados do programa de aconselhamento, em termos do custo da oferta de sessões individuais de aconselhamento, assim como a economia esperada, em relação a evitar inadimplências, em um intervalo de limites. Haverá um custo e uma economia diferentes em cada limite, porque esperamos que cada limite resulte em um número diferente de previsões positivas e em um número diferente de verdadeiros positivos dentro dessas previsões. A primeira etapa é criar

um array de possíveis limites. Usaremos 0 a 1, crescendo em incrementos de 0,01. Execute as etapas a seguir para fazer o exercício.

Nota: O código e a saída resultante deste exercício foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2PtdRyj>.

1. Crie um intervalo de limites para calcular os custos e os benefícios esperados do aconselhamento com este código:

```
thresholds = np.linspace(0, 1, 101)
```

Ele cria 101 pontos linearmente espaçados entre 0 e 1, inclusive.

Precisamos saber a possível economia alcançada com a não inadimplência. Para calculá-la, teríamos de conhecer a fatura do próximo mês. No entanto, o cliente nos informou que ela não estará disponível na hora da criação da lista de titulares de conta a serem procurados. Logo, para estimar a possível economia, usaremos o valor médio da fatura mensal mais recente de todas as contas para representar a economia de evitarmos a inadimplência para cada conta.

Algumas contas terão valores de fatura mensal maiores e outras terão valores menores. Se houver um relacionamento importante entre o valor da fatura e a probabilidade de inadimplência, usar um valor de fatura médio para calcular a economia para todas as contas pode não produzir resultados precisos. Nesse caso, um exercício de modelagem preditiva pode ser tentado para estimar a fatura do próximo mês para cada conta e assim teremos um palpite mais preciso da possível economia por conta. Porém, os valores das faturas não estavam entre as cinco características mais importantes do modelo de floresta aleatória, então, a associação com a probabilidade de inadimplência não é tão forte como com as outras características. Consequentemente, essa suposição simplificadora não deve afetar tanto os resultados.

Usaremos os dados de teste para criar essa análise, já que ela fornece uma simulação de como o modelo será empregado após ser distribuído para o cliente: com novas contas que não foram usadas no treinamento do modelo.

2. Confirme o índice do array de características de dados de teste que corresponde à fatura do mês mais recente:

```
df[features_response[:-1]].columns[5]
```

A saída deve ser:

Figura 6.34 – Nome da característica da fatura do mês mais recente.

Essa é a fatura do mês mais recente.

3. Capture o valor de fatura médio como a possível economia por inadimplência e observe-o:

```
savings_per_default = np.mean(X_test_all[:, 5])  
savings_per_default
```

A saída será:

51601.7433479286

Figura 6.35 – Média das faturas do mês mais recente entre as contas.

A média da fatura do mês mais recente entre as contas é de NT \$51,602. Logo, usando a suposição de que essa é a oportunidade de economia da não inadimplência para cada conta, a economia líquida após um custo de NT\$7,500 para aconselhamento de crédito será de NT\$44,102. Isso indica um potencial de economia líquida no programa de aconselhamento de crédito.

O problema é que nem todas as contas ficarão inadimplentes. Para uma conta que não inadimpliria, uma sessão de aconselhamento representa a perda de NT\$7,500. Nossa análise precisa balancear os custos de aconselhamento com o risco de inadimplência.

4. Armazene o custo de aconselhamento em uma variável para usar na análise:

```
cost_per_counseling = 7500
```

Também sabemos conforme informações do cliente que o programa de aconselhamento não é 100% eficaz. Devemos levar isso em consideração na nossa análise.

5. Armazene a taxa de eficácia que o cliente forneceu para usarmos na análise:

```
effectiveness = 0.70
```

Agora, calcularemos os custos e a economia para cada um dos limites. Percorreremos cada cálculo e o explicaremos, mas, por enquanto, temos de criar arrays vazios para conter os resultados de cada limite.

6. Crie arrays vazios para armazenar os resultados da análise. Explicaremos o que cada um conterà nas próximas etapas:

```
n_pos_pred = np.empty_like(thresholds)
```

```
cost_of_all_counselings = np.empty_like(thresholds)
n_true_pos = np.empty_like(thresholds)
savings_of_all_counselings = np.empty_like(thresholds)
savings_based_on_balances = np.empty_like(thresholds)
```

Essas linhas criam arrays vazios com um número de elementos igual ao dos limites de nossa análise. Percorreremos o valor de cada limite para preencher esses arrays.

7. Crie uma variável de contador e abra um loop for para percorrer os limites:

```
counter = 0
for threshold in thresholds:
```

Para cada limite haverá um número diferente de previsões positivas, de acordo com quantas probabilidades previstas estiverem acima desse limite. Elas correspondem às contas que têm previsão de inadimplir. Cada conta que tem previsão de inadimplir receberá um telefonema do aconselhamento, que tem um custo associado. Essa é a primeira parte do cálculo do custo.

8. Determine que contas têm previsões positivas neste limite:

```
pos_pred = y_test_all_predict_proba[:,1] > threshold
```

`pos_pred` é um array booleano. A soma de `pos_pred` indica o número de inadimplências previstas nesse limite.

9. Calcule o número de previsões positivas neste limite:

```
n_pos_pred[counter] = sum(pos_pred)
```

10. Calcule o custo de aconselhamento total neste limite:

```
cost_of_all_counselings[counter] = n_pos_pred[counter] * cost_per_counseling
```

Agora que caracterizamos os possíveis custos do programa de aconselhamento em cada limite, temos de ver qual é a economia possível. A economia ocorre quando o aconselhamento é oferecido para os titulares de contas certos: os que, caso contrário, ficariam inadimplentes. No caso do problema de classificação, estamos falando das previsões positivas, em que o valor real da variável de resposta também é positivo. Em outras palavras, são os verdadeiros positivos.

11. Determine que contas são verdadeiros positivos, de acordo com o array de previsões positivas e a variável de resposta:

```
true_pos = pos_pred & y_test_all.astype(bool)
```

12. Calcule o número de verdadeiros positivos como a soma do array de verdadeiros positivos:


```
n_true_pos[counter] = sum(true_pos)
```

A economia que podemos obter aconselhando com sucesso titulares de contas que, caso contrário, ficariam inadimplentes depende da economia ao evitarmos cada inadimplência, assim como da taxa de eficácia do aconselhamento. Não podemos impedir todas as inadimplências.

13. Calcule a economia antecipada nesse limite usando o número de verdadeiros positivos, a economia por inadimplência evitada e a taxa de eficácia do aconselhamento:

```
savings_of_all_counselings[counter] = n_true_pos[counter] * savings_per_default *  
effectiveness
```

14. Incremente o contador:

```
counter += 1
```

As *etapas 7 a 14* devem ser executadas como um loop for em um célula do Jupyter notebook. Depois, a economia líquida de cada limite pode ser calculada como a economia menos o custo.

15. Calcule a economia líquida para todos os limites diminuindo o array de custos do array de economia:

```
net_savings = savings_of_all_counselings - cost_of_all_counselings
```

Agora estamos prontos para visualizar quanto podemos ajudar nosso cliente a economizar ao fornecer aconselhamento para os titulares de contas apropriados. Então, vejamos.

16. Plote a economia líquida em relação aos limites:

```
mpl.rcParams['figure.dpi'] = 400  
plt.plot(thresholds, net_savings)  
plt.xlabel('Threshold')  
plt.ylabel('Net savings (NT$)')  
plt.xticks(np.linspace(0,1,11))  
plt.grid(True)
```

A plotagem resultante deve ser esta:

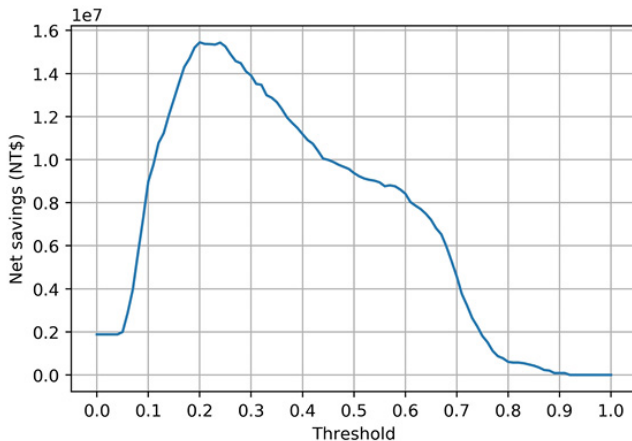


Figura 6.36 – Plote a economia líquida versus os limites.

A plotagem indica que a escolha do limite é importante. Embora seja possível gerar economia líquida para muitos valores de limite diferentes, parece que a economia líquida mais alta é gerada pela definição do limite em algum ponto do intervalo aproximado de 0.2 a 0.25.

Confirmaremos o limite ótimo para a geração da maior economia e veremos quanto economizaremos.

17. Encontre o índice do maior elemento do array de economias líquidas usando a função `argmax` do NumPy:

```
max_savings_ix = np.argmax(net_savings)
```

18. Exiba o limite que resulta na maior economia líquida:

```
thresholds[max_savings_ix]
```

A saída deve ser esta:

0.2

Figura 6.37 – Limite da maior economia líquida.

19. Exiba a maior economia líquida possível:

```
net_savings[max_savings_ix]
```

A saída será:

15446325.35991916

Figura 6.38 – Maior economia líquida.

Podemos ver que a maior economia líquida ocorre em um limite de 0.2 de probabilidade prevista de inadimplência. O valor da economia líquida obtida nesse limite é de mais de NT\$15 milhões para o dataset

de teste das contas. Essa economia teria de ser dimensionada pelo número de contas servidas pelo cliente para estimarmos a economia total possível, supondo que os dados com os quais estamos trabalhando sejam representativos de todas essas contas.

Observe, no entanto, que a economia é quase a mesma até um limite de cerca de 0.25, como visto na *Figura 6.36*.

À medida que o limite aumenta, estamos “ficando mais alertas” para o nível de risco que o cliente pode apresentar, para entrarmos em contato com eles e oferecer aconselhamento. Aumentar o limite de 0.2 para 0.25 quer dizer que só entraremos em contato com clientes de maior risco cuja probabilidade seja > 0.25 . Isso significa contatar menos clientes, reduzindo o custo direto do programa. A *Figura 6.36* indica que podemos gerar o mesmo volume de economia líquida entrando em contato com menos pessoas. Embora o efeito líquido seja o mesmo, o gasto inicial com aconselhamento será menor. Isso pode ser desejável para o cliente. Examinaremos melhor esse conceito na próxima atividade.

Atividade 6: Derivando insights financeiros

Os materiais brutos da análise financeira foram concluídos. Porém, podemos gerar alguns insights adicionais a partir desses resultados a fim de fornecer para o cliente mais contexto sobre como o modelo preditivo que construímos pode gerar valor para ele. Especificamente, examinamos os resultados para o conjunto de teste que reservamos na construção do modelo. O cliente pode ter mais contas além das fornecidas como representativas de seu negócio. Devemos relatar para ele resultados que possam ser facilmente dimensionados para qualquer que seja o tamanho de sua empresa, em termos de número de contas.

Também podemos ajudá-lo a saber quanto esse programa custará; embora a economia líquida seja um número importante a ser considerado, o cliente terá de prover fundos para o programa de aconselhamento antes de qualquer economia ser gerada. Para finalizar, ligaremos a análise financeira às métricas padrão de desempenho dos modelos de machine learning.

Quando você terminar a atividade, deve poder informar o custo inicial do programa de aconselhamento para o cliente, assim como obter plotagens de precisão e recall como esta:

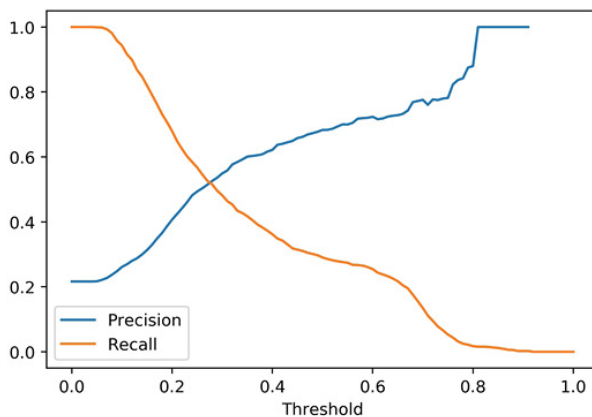


Figura 6.39 – Curva precision-recall esperada.

Essa curva será útil na interpretação do valor criado pelo modelo em diferentes limites.

Execute as etapas a seguir para concluir a atividade:

Nota: O código e a saída resultante desta atividade foram carregados em um Jupyter Notebook que pode ser encontrado aqui: <http://bit.ly/2PtdRyj>.

1. Usando o conjunto de teste, calcule o custo de todas as inadimplências se não houvesse programa de aconselhamento.
2. Calcule em que percentual o custo de inadimplências seria diminuído pelo programa de aconselhamento.
3. Calcule a economia líquida por conta no limite ótimo.
4. Plote a economia líquida por conta em relação ao custo do aconselhamento por conta para cada limite.
5. Plote a fração de contas previstas como positivas (isso se chama “flag rate”) em cada limite.
6. Plote uma curva precision-recall para os dados de teste.
7. Plote a precisão e o recall separadamente no eixo y em relação ao limite no eixo x.

Nota: A solução desta atividade consta na página XXX.

Considerações finais sobre a distribuição do modelo preditivo para o cliente

Concluimos as atividades de modelagem e também criamos uma análise financeira para indicar para o cliente como ele pode usar o modelo. Embora tenhamos criado as contribuições intelectuais básicas que são de responsabilidade dos cientistas de dados, é necessário discutir com o cliente a forma na qual todas essas contribuições serão distribuídas.

Uma contribuição importante é a capacidade preditiva incorporada ao modelo treinado. Supondo que o cliente seja capaz de trabalhar com o objeto de modelo treinado que criamos no scikit-learn, esse modelo poderia ser salvo em disco e enviado para ele. O cliente estaria então pronto para usá-lo dentro de seu. Alternativamente, pode ser necessário expressar o modelo como uma equação matemática (na regressão logística) ou como um conjunto de condições se-então (na árvore de decisão ou floresta aleatória) que o cliente possa usar para implementar a capacidade preditiva em SQL. Embora seja difícil expressar florestas aleatórias em código SQL devido à possibilidade de haver muitas árvores com vários níveis, há pacotes de software que criam essa representação automaticamente.

Antes de usar o modelo para fazer previsões, o cliente terá de *assegurar que os dados estejam preparados da mesma forma que estavam na construção de modelo que executamos*. Por exemplo, a remoção de amostras com valores 0 para todas as características, a limpeza das características EDUCATION e MARRIAGE e a imputação de PAY_1 teriam de ser feitas da maneira que demonstramos anteriormente neste capítulo. Há outras maneiras de distribuir previsões do modelo, como em um acordo em que o cliente entregue os dados para o cientista de dados e receba as previsões.

Outra consideração importante para a discussão de entrega dos produtos é: *em que formato as previsões devem ser distribuídas?* Um formato de distribuição típico para as previsões de um modelo de classificação binária, como o que criamos para o estudo de caso, é classificar as contas em sua probabilidade prevista de inadimplência. A probabilidade prevista deve ser fornecida junto com o ID da conta e qualquer outra coluna que o cliente quiser. Dessa forma, quando os funcionários do call center estiverem percorrendo a lista de titulares de contas para oferecer aconselhamento, poderão contatar os de maior risco de inadimplência primeiro e prosseguir até os titulares de contas de prioridade mais baixa conforme o tempo e os recursos permitirem. O cliente foi informado sobre o limite a ser usado para as probabilidades previstas, para que isso resulte na maior economia

líquida. Esse limite representaria o ponto de parada na lista de titulares de contas a serem contatados, se fizermos a classificação com a probabilidade prevista de inadimplência.

Para concluir, dependendo do prazo durante o qual os cientistas de dados estiverem comprometidos com o cliente, é aconselhável monitorar o desempenho do modelo com o passar do tempo, enquanto ele estiver sendo usado. A capacidade preditiva permanece a mesma ou degrada com o tempo? Ao avaliar esse quesito, é sempre importante lembrar-se de que, se os titulares das contas estiverem recebendo aconselhamento, sua probabilidade de inadimplir deve ser menor do que a probabilidade prevista indica, devido aos efeitos pretendidos com o novo programa. Portanto, e também para testar a eficácia do programa de aconselhamento, é boa prática reservar uma parcela dos titulares de contas selecionada aleatoriamente para não receber aconselhamento, independentemente do risco de inadimplência. Esse grupo é conhecido como **grupo de controle** e deve ser pequeno em relação ao resto da população que está recebendo aconselhamento, porém suficientemente grande para gerar inferências estatisticamente significativas.

Embora não faça parte do escopo deste livro examinar os detalhes de como projetar e usar um grupo de controle, é suficiente dizer que a capacidade do modelo preditivo pode ser avaliada no grupo de controle, já que seus integrantes não receberam aconselhamento, semelhante ao que ocorreu com o conjunto de contas com as quais o modelo foi treinado. Outro benefício de um grupo de controle é que a taxa de inadimplência, e a perda financeira devido às inadimplências, pode ser comparada com as contas que receberam o aconselhamento orientado pelo modelo. Se o programa estiver funcionando como esperado, as contas que estão recebendo aconselhamento terão uma taxa de inadimplência mais baixa e representarão uma perda financeira menor devido à inadimplência. O grupo de controle pode fornecer provas de que o programa está realmente funcionando.

Uma maneira relativamente simples de monitorar a implementação de um modelo é ver se a natureza das previsões está mudando com o tempo, em comparação com a população usada no treinamento do modelo. Isso pode ser feito pela plotagem das probabilidades previstas, com o uso de um histograma:

```
plt.hist(y_test_all_predict_proba[:,1], bins=30)
plt.xlabel('Predicted probability of default')
plt.ylabel('Number of accounts')
```

O gráfico da saída deve ser este:

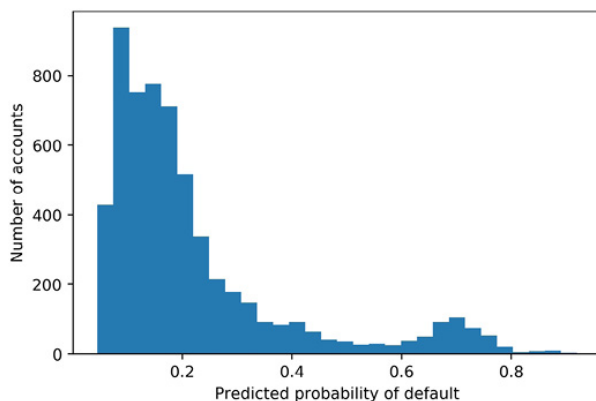


Figura 6.40 – Histograma de probabilidades previstas de inadimplência.

Se a forma do histograma de probabilidades previstas mudar substancialmente, isso pode ser um sinal de que o relacionamento entre as características e a resposta no conjunto de contas mudou e o modelo pode ter de ser retreinado ou reconstruído. Isso também pode ficar evidente se o número de contas previstas para inadimplir, de acordo com um limite selecionado, mudar de forma perceptível.

Resumo

Neste capítulo, você aprendeu a habilidade básica da ciência de dados chamada imputação. A imputação permite a substituição de valores faltantes por suposições embasadas de quais seriam os valores reais. Ela é necessária para fornecer previsões para todas as amostras, inclusive as com valores de características faltantes. Isso ocorre porque muitos algoritmos de machine learning, inclusive aqueles com os quais trabalhamos neste livro, não podem receber entradas que contenham valores faltantes. Os métodos de imputação variam de simples, como a imputação com a média, a mediana ou a moda de valores não faltantes, a mais complexos, que incluem a criação de um modelo preditivo para a característica que tem valores faltantes.

Após assegurar que o modelo que distribuiremos pode fornecer previsões para todas as amostras, por meio da imputação, conduzimos uma análise financeira. Embora tenhamos deixado isso para o fim do livro, o conhecimento dos custos e da economia trazidos pelas decisões a serem guiadas pelo modelo deve ser estimado desde o início de um projeto comum. Ele permite ao cientista de dados trabalhar

visando a um objetivo tangível em termos de aumento do lucro ou da economia. Uma etapa-chave nesse processo, para modelos de classificação binária, é selecionar um limite de probabilidade prevista no qual será declarado que uma amostra é positiva, para que os lucros ou a economia gerados pela tomada de decisões guiada pelo modelo sejam maximizados.

Para concluir, examinamos alguns pontos relacionados à distribuição e ao monitoramento do modelo, inclusive a ideia de estabelecer um grupo de controle para o monitoramento do desempenho e o teste da eficácia de programas guiados pela saída do modelo. A estrutura dos grupos de controle e as estratégias de monitoramento do modelo serão diferentes de um projeto para o outro, logo, você precisará determinar o curso de ação apropriado em cada novo caso.

Parabéns, você terminou o projeto e está pronto para distribuir suas descobertas para o cliente! Junto com os modelos treinados salvos em disco, ou outros produtos de dados ou serviços que possam ser fornecidos para o cliente, provavelmente você também vai querer criar uma apresentação, normalmente com slides, detalhando seu progresso. O conteúdo dessas apresentações costuma incluir a explicação do problema, resultados de exploração e limpeza de dados, uma comparação dos diferentes modelos construídos, qualquer outra etapa necessária como a imputação, e a análise financeira que mostra o valor do trabalho. Quando você preparar apresentações de seu trabalho, pode ser melhor *contar a história com imagens e não com muito texto*. Demonstramos muitas técnicas de visualização no decorrer do livro que podem ser usadas para isso, e você deve continuar a examinar maneiras de representar dados e resultados de modelagem.

Certifique-se sempre de perguntar ao cliente o que especificamente ele quer ver em uma apresentação e não deixe de responder a todas as suas perguntas. Quando o cliente perceber que você pode gerar valor para ele de maneira compreensível, o sucesso terá sido alcançado.

Apêndice

Esta seção foi incluída para ajudar os alunos a executar as atividades do livro. Ela contém as etapas detalhadas que devem ser executadas pelos alunos para que atinjam os objetivos das atividades.

Capítulo 1: Exploração e limpeza de dados

Atividade 1: Explorando as características financeiras restantes do dataset

1. Crie listas com nomes para as características financeiras restantes.

Essas características pertencem a dois grupos, logo, criaremos listas de nomes de características como antes para facilitar sua análise em conjunto. Você pode fazer isso com o código a seguir:

```
bill_feats = ['BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5',  
             'BILL_AMT6']  
pay_amt_feats = ['PAY_AMT1', 'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5',  
                'PAY_AMT6']
```

2. Use `.describe()` para examinar as sínteses estatísticas das características de valor da fatura. Reflita sobre o que viu. Faz sentido?

Use este código para visualizar a síntese:

```
df[bill_feats].describe()
```

A saída é mostrada na Figura A.1.

Podemos ver que a cobrança média mensal é de aproximadamente 40.000 a 50.000 dólares taiwaneses. Recomendo que o leitor examine a taxa de conversão de sua moeda local. Por exemplo, 1 dólar americano $\sim$ 30 dólares taiwaneses. Faça a conversão e pondere, é um pagamento mensal razoável? Também devemos confirmar isso com o cliente, mas parece razoável.

Observe também que há alguns valores de faturas negativos. Isso parece correto devido a um possível pagamento a mais da fatura do

mês anterior, talvez como antecipação de uma compra que apareceria na fatura do mês atual. Um cenário assim deixaria essa conta com saldo negativo, gerando um crédito para seu titular.

```
df[bill_feats].describe()
```

	BILL_AMT1	BILL_AMT2	BILL_AMT3	BILL_AMT4	BILL_AMT5	BILL_AMT6
count	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000	26664.000000
mean	51405.730723	49300.001500	47026.340047	43338.894539	40338.136701	38889.872337
std	73633.687106	70934.549534	68705.359524	64275.250740	60705.944083	59432.541657
min	-165580.000000	-69777.000000	-157264.000000	-170000.000000	-81334.000000	-339603.000000
25%	3580.000000	2999.750000	2627.250000	2341.750000	1745.000000	1256.000000
50%	22361.000000	21150.000000	20079.500000	19037.000000	18066.000000	17005.000000
75%	67649.750000	64395.500000	60360.000000	54727.500000	50290.500000	49253.750000
max	746814.000000	671563.000000	855086.000000	706864.000000	823540.000000	699944.000000

Figura A.1 – Descrição estatística dos valores de faturas dos últimos 6 meses.

3. Visualize as características de valor da fatura usando uma grade 2 por 3 de plotagens de histograma com o código a seguir:

```
df[bill_feats].hist(bins = 20, layout = (2,3))
```

O gráfico é mostrado na Figura A.2.

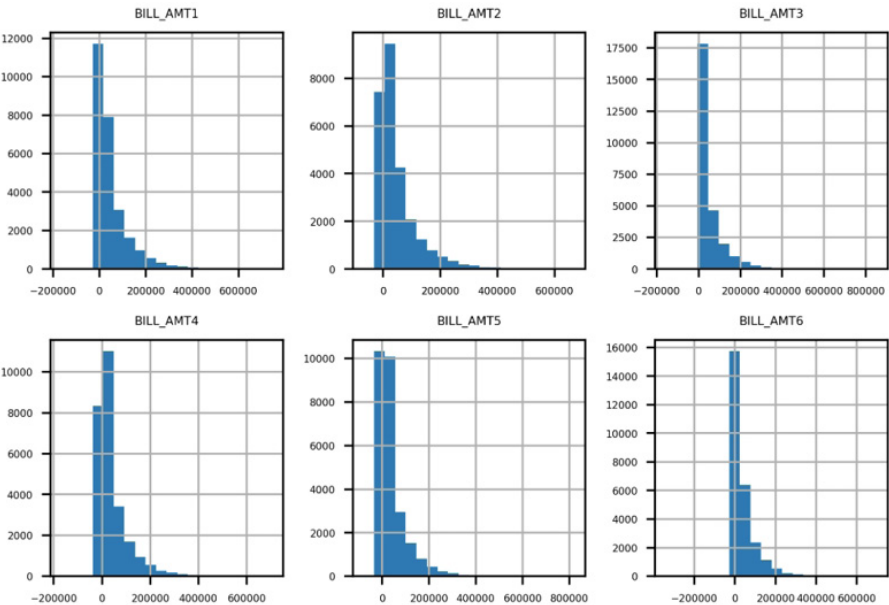


Figura A.2 – Histogramas de valores de faturas.

As plotagens de histogramas da Figura A.2 fazem sentido em vários aspectos. A maioria das contas tem faturas relativamente baixas. Há uma diminuição estável no número de contas à medida que o valor

da fatura aumenta. Também parece que a distribuição de pagamentos é mais ou menos semelhante de um mês para o outro, logo, não notamos nenhum problema de inconsistência de dados como nas características de status do pagamento. Essa característica parece passar em nossa inspeção de qualidade dos dados. Agora seguiremos para o conjunto final de características.

4. Use o método `.describe()` para obter um resumo das características de valor do pagamento usando o código a seguir:

```
df[pay_amt_feats].describe()
```

A saída deve ser esta:

```
df[pay_amt_feats].describe()
```

	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AMT5	PAY_AMT6
count	26664.000000	2.666400e+04	26664.000000	26664.000000	26664.000000	26664.000000
mean	5704.085771	5.881110e+03	5259.514964	4887.048717	4843.729973	5257.843047
std	16699.398632	2.121431e+04	17265.439561	15956.349371	15311.721795	17635.468185
min	0.000000	0.000000e+00	0.000000	0.000000	0.000000	0.000000
25%	1000.000000	8.020000e+02	390.000000	294.750000	242.750000	111.000000
50%	2114.500000	2.007000e+03	1822.000000	1500.000000	1500.000000	1500.000000
75%	5027.000000	5.000000e+03	4556.250000	4050.500000	4082.750000	4015.000000
max	873552.000000	1.227082e+06	889043.000000	621000.000000	426529.000000	528666.000000

Figura A.3 – Descrição estatística dos valores de pagamentos de faturas nos últimos 6 meses.

Os valores de pagamento médios têm uma ordem de magnitude (potência de 10) menor que os valores de fatura médios que resumimos na Atividade. Isso significa que o “caso médio” é uma conta que não está liquidando seu saldo total de um mês para o outro. Isso faz sentido de acordo com nossa exploração da característica `PAY_1`, para a qual o valor predominante era 0 (a conta fez o pagamento mínimo, mas não liquidou o saldo total). Não há pagamentos negativos, o que também parece correto.

5. Plote um histograma das características de pagamento da fatura semelhante ao das características de valor da fatura, mas aplique também alguma rotação aos rótulos do eixo *x* com o argumento de palavra-chave `xrot` para que eles não se sobreponham. Use o argumento de palavra-chave `xrot = <ângulo>` para girar os rótulos do eixo *x* de acordo com um ângulo específico em graus com o código a seguir:

```
df[pay_amt_feats].hist(layout=(2,3), xrot=30)
```

Em nosso caso, vimos que 30 graus de rotação funcionaram bem. A plotagem deve ser esta:

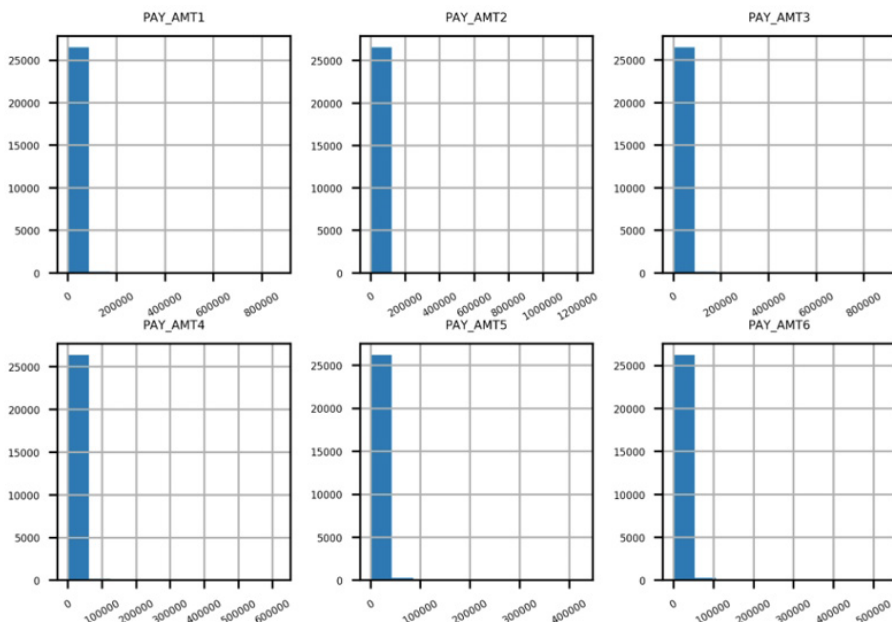


Figura A.4 – Histogramas de dados brutos de valores de pagamento.

Uma olhada rápida nessa figura mostra que esse não é um gráfico muito informativo; há apenas um bin na maioria dos histogramas que tem uma altura significativa. Essa não é uma maneira eficaz de visualizar esses dados. Parece que os valores de pagamentos mensais estão principalmente em um bin que inclui 0. Quantos são de fato 0?

6. Use uma máscara booleana para ver quantos dos dados de valor de pagamento são exatamente iguais a 0 com o código a seguir:

```
pay_zero_mask = df[pay_amt_feats] == 0  
pay_zero_mask.sum()
```

A saída será:

```
pay_zero_mask = df[pay_amt_feats] == 0  
  
pay_zero_mask.sum()  
  
PAY_AMT1    4656  
PAY_AMT2    4833  
PAY_AMT3    5293  
PAY_AMT4    5697  
PAY_AMT5    5981  
PAY_AMT6    6373  
dtype: int64
```

Figura A.5 – Contagens de pagamentos de fatura iguais a 0.

Esses dados fazem sentido de acordo com o histograma da etapa anterior?

A primeira linha cria um novo DataFrame chamado `pay_zero_mask`, que é um DataFrame de valores `True` e `False` de acordo com se o valor do pagamento é igual a 0. A segunda linha calcula as somas das colunas desse DataFrame, interpretando `True` como 1 e `False` como 0, para que as somas indiquem quantas contas têm valor 0 para cada característica.

Podemos ver que uma parte substancial das contas, cerca de 20-25%, tem pagamento da fatura igual a 0 em algum mês específico. No entanto, a maioria dos pagamentos está acima de 0. Por que não os vemos no histograma? Porque o **intervalo** de valores corresponde aos valores da maioria dos pagamentos.

Na síntese estatística, podemos ver que o pagamento máximo em um mês normalmente é 2 ordens de magnitude (100 vezes) maior que o pagamento médio. Parece que há apenas um pequeno número desses pagamentos muito altos. No entanto, devido à maneira como o histograma foi criado, usando bins de tamanho igual, quase todos os dados estão amontoados no bin menor, e os bins maiores são quase invisíveis porque têm poucas contas. Precisamos de uma estratégia para visualizar esses dados com mais eficácia.

7. Ignorando os pagamentos iguais a 0 usando a máscara que criou na etapa anterior, utilize o método `.apply()` do pandas e o método `np.log10()` do NumPy para plotar histogramas de transformações logarítmicas dos pagamentos diferentes de zero. Você pode usar `.apply()` para aplicar uma função, inclusive `log10`, a todos os elementos de um DataFrame. Use o código a seguir para concluir a etapa anterior:

```
df[pay_amt_feats][~pay_zero_mask].apply(np.log10).hist(layout=(2,3))
```

Esse é um uso relativamente avançado do pandas, então, não se preocupe se não conseguir entendê-lo. Mesmo assim, é um bom começo ver que é possível fazer muita coisa com o pandas com pouco código.

A saída deve ser:

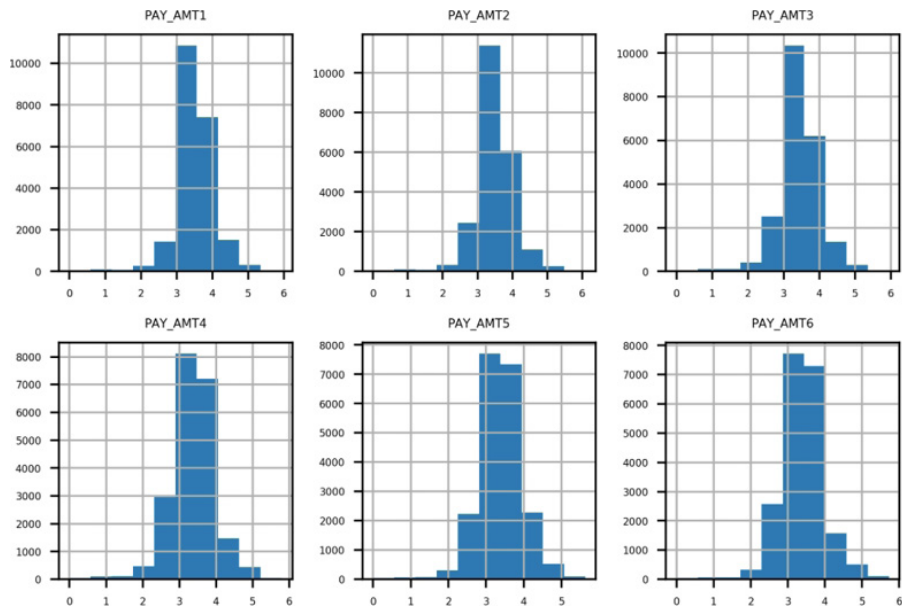


Figura A.6 – Logs de base 10 de valores de pagamento de fatura diferentes de zero.

Embora pudéssemos ter tentado criar bins de largura variável para uma melhor visualização dos valores de pagamento, uma abordagem mais conveniente muito usada na visualização, e às vezes até na modelagem, de dados que têm alguns valores em uma escala muito diferente da maioria deles é a transformação logarítmica, ou **log transform**. Usamos uma log transform de base 10. Essa transformação informa o número de zeros de um valor. Em outras palavras, um saldo de um milhão de dólares teria uma log transform de pelo menos 6, mas menos de 7, porque $10^6 = 1.000.000$ (e inversamente $\log_{10}(1.000.000) = 6$) enquanto $10^7 = 10.000.000$.

Para aplicar essa transformação aos nossos dados, primeiro tivemos de mascarar os pagamentos iguais a zero, porque $\log_{10}(0)$ é indefinido. Fizemos isso com o operador lógico `not ~` de Python e a máscara de zeros que criamos. Em seguida, empregamos o método `.apply()` do pandas, que aplica qualquer função aos dados selecionados. Nesse caso, quisemos aplicar um logaritmo de base 10, calculado por `np.log10`. Para concluir, criamos histogramas desses valores.

O resultado é uma visualização de dados um pouco mais eficaz: os valores estão distribuídos de maneira mais informativa pelos bins do histograma. Podemos ver que os pagamentos que ocorrem mais

estão no intervalo dos milhares ($\log_{10}(1.000) = 3$), o que coincide com o que observamos para o pagamento médio na síntese estatística. Há alguns pagamentos muito baixos, e também alguns muito altos. No geral, a distribuição de pagamentos parece bem consistente de um mês para o outro, logo, não estamos vendo nenhum possível problema com esses dados.

Capítulo 2: Introdução ao Scikit-Learn e avaliação do modelo

Atividade 2: Executando a regressão logística com uma nova característica e criando uma curva precision-recall

1. Use o método `train_test_split` do scikit-learn para criar um novo conjunto de dados de treinamento e de teste. Dessa vez, em vez de EDUCATION, use LIMIT_BAL: o limite de crédito da conta.

Execute o código a seguir para fazê-lo:

```
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(
    df[LIMIT_BAL].values.reshape(-1,1), df[default payment next month'].values,
    test_size=0.2, random_state=24)
```

Observe que criamos novas divisões de treinamento e teste, com novos nomes de variáveis.

2. Treine um modelo de regressão logística usando os dados de treinamento provenientes da divisão.

O código a seguir faz isso:

```
example_lr.fit(X_train_2, y_train_2)
```

Reutilizamos o mesmo objeto de modelo, `example_lr`. Podemos **retreinar** esse objeto para que aprenda o relacionamento entre essa nova característica e a resposta. Poderíamos até mesmo testar uma divisão de treinamento/teste diferente, se quiséssemos, sem criar um novo objeto de modelo. O objeto de modelo existente foi atualizado **in loco**.

3. Crie o array de probabilidades previstas para os dados de teste:

Aqui está o código dessa etapa:

```
y_test_2_pred_proba = example_lr.predict_proba(X_test_2)
```

4. Calcule a ROC AUC usando as probabilidades previstas e os rótulos verdadeiros dos dados de teste. Compare o resultado com a ROC

AUC da característica EDUCATION.

Execute este código para essa etapa:

```
metrics.roc_auc_score(y_test_2, y_test_2_pred_proba[:,1])
```

A saída é:

```
metrics.roc_auc_score(y_test_2, y_test_2_pred_proba[:,1])  
0.6201990844642832
```

Figura A.7 – Calculando a ROC AUC.

Observe que indexamos o array de probabilidades previstas para obter a da classe positiva da segunda coluna. O que podemos ver se compararmos com a ROC AUC da regressão logística EDUCATION? A AUC é mais alta. Isso pode estar ocorrendo porque agora estamos usando uma característica que tem algo a ver com o status financeiro da conta (limite de crédito) para prever outra coisa relacionada ao status financeiro (se ela vai ou não ficar inadimplente), em vez de usar algo menos diretamente relacionado a finanças.

5. Plote a curva ROC.

Aqui está o código que faz isso; é semelhante ao código que usamos no exercício anterior:

```
fpr_2, tpr_2, thresholds_2 = metrics.roc_curve(y_test_2, y_test_2_pred_proba[:,1])  
plt.plot(fpr_2, tpr_2, '*-')  
plt.plot([0, 1], [0, 1], 'r--')  
plt.legend(['Logistic regression', 'Random chance'])  
plt.xlabel('FPR')  
plt.ylabel('TPR')  
plt.title('ROC curve for logistic regression with LIMIT_BAL feature')
```

A plotagem é mostrada na Figura A.8.

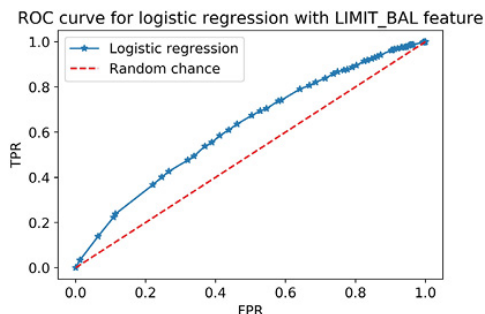


Figura A.8 – Curva ROC da regressão logística de LIMIT_BAL.

Isso parece mais com a curva ROC que gostaríamos de ver: está

mais distante da linha de chance aleatória que o modelo que usa apenas EDUCATION. Observe também que a variação nos pares de taxas de verdadeiros e falsos positivos é um pouco mais suave no intervalo de limites, o que reflete o maior número de diferentes valores da característica LIMIT_BAL.

6. Calcule os dados da curva precision-recall baseando-se nos dados de teste e usando a funcionalidade do scikit-learn.

Com frequência a precisão é considerada junto com o recall. Você já está familiarizado com o recall. É apenas outra palavra para a taxa de verdadeiros positivos. Podemos usar `precision_recall_curve` em `sklearn.metrics` para variar automaticamente o limite e calcular pares de valores de precisão e recall em cada um. Aqui está o código que recupera esses valores, que é semelhante a `roc_curve`:

```
precision, recall, thresh_3 = \
metrics.precision_recall_curve(y_test_2, y_test_2_pred_proba[:,1])
```

7. Plote a curva precision-recall usando o matplotlib: podemos fazer isso com o código a seguir.

Observe que inserimos o recall no eixo x, a precisão no eixo y, e configuramos os limites dos eixos com o intervalo [0, 1]:

```
plt.plot(recall, precision, '-x')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Another logistic regression with just one feature: LIMIT_BAL')
plt.xlim([0, 1])
plt.ylim([0, 1])
```

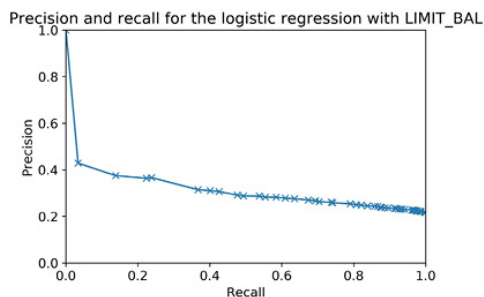


Figura A.9 – Plotagem da curva precision-recall.

8. Use o scikit-learn para calcular a área sob a curva precision-recall.

Aqui está o código que faz isso:

```
metrics.auc(recall, precision)
```

Você obterá a seguinte saída:

```
metrics.auc(recall, precision)
```

```
0.31566964427378624
```

Figura A.10 – Área sob a curva precision-recall.

Vimos que a curva precision-recall mostra que geralmente a precisão é bastante baixa para esse modelo; para quase todo o intervalo de limites, a precisão, ou a parte das classificações positivas que está correta, é menos da metade. Podemos calcular a área sob a curva precision-recall como uma maneira de comparar esse classificador com outros modelos ou conjuntos de características que possamos vir a considerar.

O scikit-learn oferece a funcionalidade de cálculo de uma área sob a curva de qualquer conjunto de dados x-y, usando a regra trapezoidal, da qual você deve se lembrar pela integral: `metrics.auc`. Usamos essa funcionalidade para obter a área sob a curva precision-recall.

9. Agora recalcule a ROC AUC, mas dessa vez faça-o para os dados de treinamento. Em que esse cálculo é diferente, conceitualmente e quantitativamente, do que fizemos antes?

Primeiro, temos de calcular as probabilidades previstas usando os dados de treinamento em vez dos dados de teste. Em seguida, calcularemos a ROC AUC usando os rótulos dos dados de treinamento. Aqui está o código:

```
y_train_2_pred_proba = example_lr.predict_proba(X_train_2)
metrics.roc_auc_score(y_train_2, y_train_2_pred_proba[:,1])
```

Você deve obter a seguinte saída:

```
y_train_2_pred_proba = example_lr.predict_proba(X_train_2)
```

```
metrics.roc_auc_score(y_train_2, y_train_2_pred_proba[:,1])
```

```
0.6182918113358344
```

Figura A.11 – ROC AUC dos dados de treinamento.

Quantitativamente, podemos ver que essa AUC não é tão diferente da ROC AUC dos dados de teste que calculamos anteriormente. As duas são aproximadamente iguais a 0.62. Conceitualmente qual é a diferença? Quando calculamos essa métrica com os dados de treinamento, estamos medindo a habilidade do modelo de prever os mesmos dados que o “ensinaram” como fazer previsões. Estamos vendo *com que eficácia o modelo ajusta os dados*. Se compararmos isso com uma métrica de dados de teste, estaremos comparando **pontuações de treinamento e teste**. Se houver alguma diferença

nessa pontuação, que geralmente ocorre na forma de uma pontuação de treinamento mais alta que a de teste, isso indicará que, embora o modelo ajuste bem os dados, o modelo treinado não pode ser generalizado com bom resultado para novos dados desconhecidos.

Nesse caso, as pontuações de treinamento e teste são semelhantes, significando que o modelo se sai bem com dados fora da amostra assim como com os mesmos dados usados no seu treinamento. Você pode aprender mais sobre os insights que podemos obter comparando as pontuações de treinamento e teste no *Capítulo 4, O trade-off entre viés e variância*.

Capítulo 3: Detalhes da regressão logística e exploração de características

Atividade 3: Ajustando um modelo de regressão logística e usando os coeficientes diretamente

As primeiras etapas são semelhantes a operações que executamos em atividades anteriores:

1. Crie uma divisão treinamento/teste (80/20) com PAY_1 e LIMIT_BAL como características:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    df[['PAY_1', 'LIMIT_BAL']].values, df['default payment next month'].values,
    test_size=0.2, random_state=24)
```

2. Importe LogisticRegression com as opções padrão, mas configure o solver com 'liblinear'.

```
from sklearn.linear_model import LogisticRegression
lr_model = LogisticRegression(solver='liblinear')
```

3. Treine o modelo com os dados de treinamento e obtenha as classes previstas, assim como as probabilidades das classes, usando os dados de teste:

```
lr_model.fit(X_train, y_train)
y_pred = lr_model.predict(X_test)
y_pred_proba = lr_model.predict_proba(X_test)
```

4. Extraia os coeficientes e a interceptação do modelo treinado e calcule manualmente as probabilidades previstas. Você terá de adicionar uma coluna de 1's às suas características para multiplicar pela interceptação.

Primeiro, criaremos o array de características, com uma coluna de 1's adicionada, usando o empilhamento horizontal:

```
ones_and_features = np.hstack([np.ones((X_test.shape[0],1)), X_test])
```

Agora precisamos da interceptação e dos coeficientes, que redimensionaremos e concatenaremos a partir da saída do scikit-learn:

```
intercept_and_coefs = np.concatenate([lr_model.intercept_.reshape(1,1), lr_model.coef_],  
axis=1)
```

Para multiplicar repetidamente a interceptação e os coeficientes por todas as linhas de ones_and_features, e calcular a soma de cada linha (isto é, encontrar a combinação linear), você poderia escrever tudo isso usando a multiplicação e a adição. No entanto, é muito mais rápido usar a função de produto dot:

```
X_lin_comb = np.dot(intercept_and_coefs, np.transpose(ones_and_features))
```

Agora X_lin_comb tem o argumento que precisamos passar para a função sigmóide que definimos para calcular as probabilidades previstas:

```
y_pred_proba_manual = sigmoid(X_lin_comb)
```

5. Usando um limite de 0.5, calcule manualmente as classes previstas. Compare o resultado com a saída de previsões de classes do scikit-learn.

As probabilidades previstas manualmente, y_pred_proba_manual, devem ser as mesmas de y_pred_proba; verificaremos isso em breve. Primeiro, preveja manualmente as classes com o limite:

```
y_pred_manual = y_pred_proba_manual >= 0.5
```

Esse array terá uma dimensão diferente de y_pred, mas deve conter os mesmos valores. Podemos verificar se todos os elementos dos dois arrays são iguais desta forma:

```
np.array_equal(y_pred.reshape(1,-1), y_pred_manual)
```

True

Figura A.12 – Igualdade de arrays do NumPy.

6. Calcule a ROC AUC usando as probabilidades previstas pelo scikit-learn e as que você calculou manualmente e compare.

Primeiro, importe o seguinte:

```
from sklearn.metrics import roc_auc_score
```

Em seguida, calcule essa métrica nas duas versões, tomando o cuidado de acessar a coluna correta, ou redimensione quando

necessário:

```
roc_auc_score(y_test, y_pred_proba_manual.reshape(y_pred_proba_manual.shape[1],))  
0.627207450280691
```

```
roc_auc_score(y_test, y_pred_proba[:,1])  
0.627207450280691
```

Figura A.13 – Calculando as ROC AUCs a partir das probabilidades previstas.

As AUCs são iguais. O que fizemos aqui? Confirmamos que tudo de que precisamos nesse modelo ajustado do scikit-learn são três números: a interceptação e os dois coeficientes. Uma vez que os tivermos, poderemos criar previsões do modelo usando algumas linhas de código, com funções matemáticas, que sejam equivalentes às previsões feitas diretamente a partir do scikit-learn.

Isso é bom para confirmar o que sabemos, mas, caso contrário, por que o faríamos? Falaremos sobre a **implantação do modelo** no último capítulo. No entanto, dependendo das circunstâncias, você pode se ver em uma situação em que não tenha acesso ao Python em um ambiente em que novas características tenham de ser fornecidas ao modelo para previsão. Por exemplo, você pode ter de fazer previsões totalmente em SQL. Embora isso seja uma limitação, com a regressão logística você pode usar as funções matemáticas que estão disponíveis em SQL para recriar a previsão, tendo apenas de copiar e colar a interceptação e os coeficientes em algum local de seu código SQL. O produto dot pode não estar disponível, mas você pode usar a multiplicação e a adição para atingir o mesmo objetivo.

E quanto aos resultados propriamente ditos? O que vimos aqui é que podemos melhorar um pouco o desempenho do modelo além do que já fizemos: usando apenas LIMIT_BAL como característica na Atividade do capítulo anterior, a ROC AUC foi um pouco menor em 0.62, em vez de em 0.63 como no caso atual. No próximo capítulo, aprenderemos técnicas avançadas com a regressão logística que poderemos usar para aumentar o desempenho mais do que aumentamos aqui.

Capítulo 4: O trade-off entre viés e variância

Atividade 4: Validação cruzada e engenharia de características com os dados do estudo de caso

1. Selecione as características no DataFrame dos dados do estudo de caso.

Você pode usar a lista de nomes de características que criamos neste capítulo. No entanto, certifique-se de não incluir a variável de resposta, que seria uma característica muito boa (mas totalmente inapropriada):

```
features = features_response[:-1]
X = df[features].values
```

2. Crie uma divisão de treinamento/teste usando um seed aleatório igual a 24:

```
X_train, X_test, y_train, y_test = train_test_split(X, df['default payment next month'].values,
test_size=0.2, random_state=24)
```

Usaremos essa divisão ao avançar e reservaremos os dados de teste como o conjunto de teste desconhecido. Dessa forma, poderemos criar facilmente notebooks separados com outras abordagens de modelagem, usando os mesmos dados de treinamento.

3. Instancie MinMaxScaler para escalonar os dados, como mostrado no código a seguir:

```
from sklearn.preprocessing import MinMaxScaler
min_max_sc = MinMaxScaler()
```

4. Instancie um modelo de regressão logística com o solver saga e a penalização L1, e configure max_iter com 1.000, já que queremos que o solver tenha iterações suficientes para encontrar uma boa solução:

```
lr = LogisticRegression(solver='saga', penalty='l1', max_iter=1000)
```

5. Importe a classe Pipeline e crie um Pipeline com o escalonador e o modelo de regressão logística, usando os nomes 'scaler' e 'model' para as etapas, respectivamente:

```
from sklearn.pipeline import Pipeline
scale_lr_pipeline = Pipeline(steps=[('scaler', min_max_sc), ('model', lr)])
```

6. Use os métodos get_params e set_params para visualizar os parâmetros de cada estágio do pipeline e alterá-los:

```
scale_lr_pipeline.get_params()
scale_lr_pipeline.get_params()['model__C']
scale_lr_pipeline.set_params(model__C = 2)
```

7. Crie um intervalo menor de valores de C para testar com a validação cruzada, já que esses modelos demandarão mais tempo para serem treinados e testados com mais dados do que nossos

exercícios anteriores; recomendamos $C = [10^2, 10, 1, 10^{-1}, 10^{-2}, 10^{-3}]$:

```
C_val_exponents = np.linspace(2,-3,6)
C_vals = np.float(10)**C_val_exponents
```

8. Crie uma nova versão da função `cross_val_C_search`, chamada `cross_val_C_search_pipe`. Em vez do argumento `model`, essa função receberá um argumento `pipeline`. As alterações na função serão a definição do valor de C com o uso de `set_params(model_C = <valor que você quiser testar>)` no `pipeline`, a substituição do modelo pelo `pipeline` para os métodos `fit` e `predict_proba` e o acesso ao valor de C com o uso de `pipeline.get_params()['model_C']` para a atualização do status exibido.

As alterações são as seguintes:

```
def cross_val_C_search_pipe(k_folds, C_vals, pipeline, X, Y):
    ##[...]
    pipeline.set_params(model_C = C_vals[c_val_counter])
    ##[...]
    pipeline.fit(X_cv_train, y_cv_train)
    ##[...]
    y_cv_train_predict_proba = pipeline.predict_proba(X_cv_train)
    ##[...]
    y_cv_test_predict_proba = pipeline.predict_proba(X_cv_test)
    ##[...]
    print('Done with C = {}'.format(pipeline.get_params()['model_C']))
```

Nota: Para ver o código completo, acesse <http://bit.ly/2ZAy2Pr>.

9. Execute essa função como no exercício anterior, mas usando o novo intervalo de valores de C , o `pipeline` que você criou e as características e a variável de resposta da divisão de treinamento dos dados do estudo de caso. Você pode ver avisos aqui, ou em etapas posteriores, sobre a não convergência do `solver`; poderíamos tentar usar as opções `tol` ou `max_iter` para chegar à convergência, mas os resultados obtidos com `max_iter = 1000` devem ser suficientes. Aqui está o código que faz isso:

```
cv_train_roc_auc, cv_test_roc_auc, cv_test_roc = \
cross_val_C_search_pipe(k_folds, C_vals, scale_lr_pipeline, X_train, y_train)
```

Você obterá a seguinte saída:

```
Done with C = 100.0
Done with C = 10.0
Done with C = 1.0
Done with C = 0.1
Done with C = 0.01
```

10. Plote a ROC AUC média de treinamento e teste entre os folds para cada valor de C usando o código a seguir:

```
plt.plot(C_val_exponents, np.mean(cv_train_roc_auc, axis=0), '-o',
        label='Average training score')
plt.plot(C_val_exponents, np.mean(cv_test_roc_auc, axis=0), '-x',
        label='Average testing score')
plt.ylabel('ROC AUC')
plt.xlabel('log10(C)')
plt.legend()
plt.title('Cross validation on Case Study problem')
np.mean(cv_test_roc_auc, axis=0)
```

Você obterá a saída mostrada na Figura A.14.

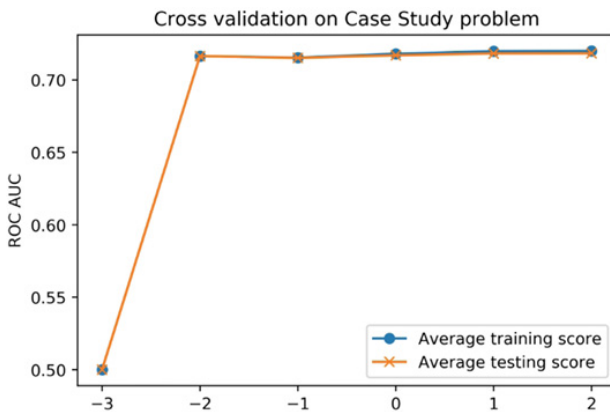


Figura A.14 – Desempenho de teste da validação cruzada.

Você deve ter notado que a regularização não apresenta muitos benefícios aqui, como era de se esperar. Embora seja possível melhorar o desempenho do modelo para além de nossos esforços anteriores usando todas as características disponíveis, parece que não há ocorrência de overfitting. Em vez disso, as pontuações de treinamento e teste são quase iguais. Em vez de overfitting, é possível que ocorra underfitting. Tentaremos criar algumas características de interação por engenharia para ver se elas podem melhorar o desempenho.

11. Crie características de interação para os dados do estudo de caso e confirme se o número de características novas faz sentido usando o código a seguir:

```
from sklearn.preprocessing import PolynomialFeatures
make_interactions = PolynomialFeatures(degree=2, interaction_only=True,
                                       include_bias=False)
X_interact = make_interactions.fit_transform(X)
```



```
X_train, X_test, y_train, y_test = train_test_split(
X_interact, df['default payment next month'].values,
test_size=0.2, random_state=24)
print(X_train.shape)
print(X_test.shape)
```

Você obterá esta saída:

```
(21331, 153)
```

```
(5333, 153)
```

Nela podemos ver que o número de características é 153, que é $17 + "17 \text{ choose } 2" = 17 + 136 = 153$. A parte " $17 \text{ choose } 2$ " vem da seleção de todas as combinações possíveis de 2 características para interagir a partir das 17 características possíveis.

12. Repita o procedimento de validação cruzada e observe o desempenho do modelo agora; isto é, repita as *etapas 9 e 10*. Observe que levaremos mais tempo devido ao número maior de características, mas provavelmente serão apenas alguns minutos.

Obteremos a seguinte saída:

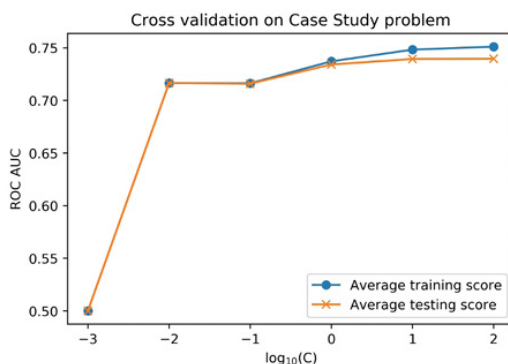


Figura A.15 – Melhora no desempenho de teste da validação cruzada pela inclusão de características de interação.

O desempenho médio de teste da validação cruzada melhorou com as características de interação? A regularização foi útil?

A criação de características de interação por engenharia aumenta a melhor pontuação de teste do modelo para uma $ROC AUC = 0.74$ na média entre os folds, que sem a inclusão das interações era de 0.72. Essas pontuações ocorrem em $C = 100$, isto é, com uma regularização insignificante. Na plotagem de pontuações de treinamento versus teste para o modelo com interações, vemos que a pontuação de treinamento é um pouco mais alta que a de teste, logo, poderíamos dizer que algum overfitting está ocorrendo. No entanto, não podemos aumentar a pontuação de teste pela regularização aqui, então, talvez esse não

seja um caso problemático de overfitting. Quase sempre, a estratégia que produz a pontuação de teste mais alta é a melhor.

Guardaremos a etapa de ajuste com todos os dados de treinamento para depois, quando tivermos testado outros modelos na validação cruzada para encontrar o melhor.

Capítulo 5: Árvores de decisão e florestas aleatórias

Atividade 5: Busca em grade na validação cruzada com floresta aleatória

1. Crie um dicionário representando a grade para os hiperparâmetros `max_depth` e `n_estimators` que serão pesquisados. Inclua as profundidades 3, 6, 9 e 12, e 10, 50, 100 e 200 árvores. Deixe os outros hiperparâmetros com seus padrões. Crie o dicionário usando este código:

```
rf_params = {'max_depth':[3, 6, 9, 12],  
             'n_estimators':[10, 50, 100, 200]}
```

Nota: Há muitos outros hiperparâmetros que poderíamos pesquisar. Especificamente, a documentação do scikit-learn para a floresta aleatória indica o seguinte:

“Os principais parâmetros a serem ajustados com o uso desses métodos são `n_estimators` e `max_features`” e “Bons valores padrão empíricos são ... `max_features=sqrt(n_features)` para tarefas de classificação”.

Fonte: <https://scikit-learn.org/stable/modules/ensemble.html#parameters>

Para os fins deste livro, usaremos `max_features='auto'` (que é igual a `sqrt(n_features)`) e limitaremos nossa exploração a `max_depth` e `n_estimators` para termos um runtime mais curto. Em uma situação do mundo real, você deve explorar outros hiperparâmetros de acordo com o tempo de computação que tiver. Lembre-se de que, para pesquisar em espaços de parâmetros especialmente grandes, você pode usar `RandomizedSearchCV` a fim de evitar calcular exaustivamente métricas para cada combinação de hiperparâmetros na grade.

2. Instancie um objeto `GridSearchCV` usando as mesmas opções que empregamos anteriormente neste capítulo, mas com o dicionário de hiperparâmetros criado na etapa 1 desta atividade. Configure

verbose=2 para ver a saída de cada ajuste executado. Você pode reutilizar o mesmo objeto de modelo de floresta aleatória rf que estivemos usando. Instancie a classe usando este código:

```
cv_rf = GridSearchCV(rf, param_grid=rf_params, scoring='roc_auc', fit_params=None,
                    n_jobs=None, iid=False, refit=True, cv=4, verbose=2,
                    pre_dispatch=None, error_score=np.nan, return_train_score=True)
```

3. Ajuste o objeto GridSearchCV com os dados de treinamento. Faça a busca na grade usando este código:

```
cv_rf.fit(X_train, y_train)
```

Já que selecionamos a opção verbose=2, você verá um volume relativamente grande de saídas no notebook. Haverá uma saída para cada combinação de hiperparâmetros e para cada fold, quando ele for ajustado e testado. As primeiras linhas da saída são mostradas na Figura A.16.

Embora não seja necessário ver toda essa saída para procedimentos de validação cruzada mais curtos, para os mais longos pode ser tranquilizador saber que a validação cruzada está funcionando e para sabermos quanto o ajuste está demorando para várias combinações de hiperparâmetros. Se estiver demorando muito, pode ser melhor interromper o kernel pressionando o botão de parada (quadrado) no topo do notebook e selecionar hiperparâmetros que demorem menos tempo para serem executados ou usar um conjunto de hiperparâmetros mais limitado.

```
Fitting 4 folds for each of 16 candidates, totalling 64 fits
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.
[Parallel(n_jobs=1)]: Done 1 out of 1 | elapsed: 0.1s remaining: 0.0s

[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=10 .....
[CV] ..... max_depth=3, n_estimators=10, total= 0.1s
[CV] max_depth=3, n_estimators=50 .....
[CV] ..... max_depth=3, n_estimators=50, total= 0.5s
```

Figura A.16 – Saída verbosa da validação cruzada.

Quando tudo terminar, você deve ver a saída mostrada na Figura A.17.

Essa tarefa de validação cruzada levou cerca de dois minutos para ser executada. À medida que suas tarefas aumentarem, pode ser melhor usar um processamento paralelo com o parâmetro n_jobs para ver se é possível acelerar a busca. Usando n_jobs=-1 e

omitindo a opção `pre_dispatch` para que o padrão seja usado, pudemos chegar a um run-time de 52 segundos, em vez do runtime de 2 minutos do processamento serial como mostrado aqui. No entanto, com o processamento paralelo, você não poderá ver a saída de cada operação de ajuste de modelo como mostra a *Figura 5.30*.

```
[Parallel(n_jobs=1)]: Done 64 out of 64 | elapsed: 2.0min finished

GridSearchCV(cv=4, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
             max_depth=5, max_features='auto', max_leaf_nodes=None,
             min_impurity_decrease=0.0, min_impurity_split=None,
             min_samples_leaf=1, min_samples_split=2,
             min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
             oob_score=False, random_state=1, verbose=0, warm_start=False),
             fit_params=None, iid=False, n_jobs=None,
             param_grid={'max_depth': [3, 6, 9, 12], 'n_estimators': [10, 50, 100, 200]},
             pre_dispatch=None, refit=True, return_train_score=True,
             scoring='roc_auc', verbose=2)
```

Figura A.17 – Saída da validação cruzada após a conclusão.

4. Insira os resultados da busca em grade em um DataFrame do pandas. Use este código para inserir os resultados em um DataFrame:

```
cv_rf_results_df = pd.DataFrame(cv_rf.cv_results_)
```

5. Crie uma visualização com `pcolormesh` da pontuação média de teste de cada combinação de hiperparâmetros. O código a seguir cria um gráfico de malha com os resultados da validação cruzada. Ele é semelhante ao exemplo de gráfico que criamos anteriormente, mas com anotações que são específicas da validação cruzada que executamos aqui:

```
xx_rf, yy_rf = np.meshgrid(range(5), range(5))
cm_rf = plt.cm.jet
ax_rf = plt.axes()
pcolor_graph = ax_rf.pcolormesh(xx_rf, yy_rf,
                                cv_rf_results_df['mean_test_score'].values.reshape((4,4)), cmap=cm_rf)
plt.colorbar(pcolor_graph, label='Average testing ROC AUC')
ax_rf.set_aspect('equal')
ax_rf.set_xticks([0.5, 1.5, 2.5, 3.5])
ax_rf.set_yticks([0.5, 1.5, 2.5, 3.5])
ax_rf.set_xticklabels([str(tick_label) for tick_label in rf_params['n_estimators']])
ax_rf.set_yticklabels([str(tick_label) for tick_label in rf_params['max_depth']])
ax_rf.set_xlabel('Number of trees')
ax_rf.set_ylabel('Maximum depth')
```

As principais alterações que fizemos em relação ao exemplo anterior são que, em vez de plotar os inteiros de 1 a 16, estamos plotando as pontuações médias de teste que recuperamos e redimensionamos com

`cv_rf_results_df['mean_test_score'].values.reshape((4,4)).` O outro elemento novo é termos usado list comprehensions para criar listas de strings para tick labels, com base nos valores numéricos de hiperparâmetros da grade. Eles foram acessados no dicionário que definimos e então convertidos individualmente para o tipo de dado str (string) dentro da list comprehension, por exemplo: `ax_rf.set_xticklabels([str(tick_label) for tick_label in rf_params['n_estimators'])]`. Já tínhamos definido os locais em que queremos que os ticks estejam usando `set_xticks`. O gráfico deve ficar com esta aparência:

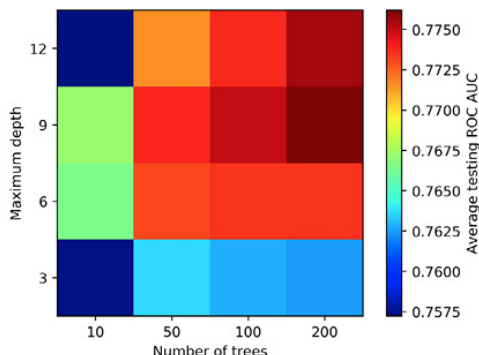


Figura A.18 – Resultados da validação cruzada de uma floresta aleatória em uma grade com dois hiperparâmetros.

6. Decida que conjunto de hiperparâmetros usará.

O que podemos concluir de nossa busca em grade? Sem dúvida parece haver uma vantagem no uso de árvores com profundidade maior que 3. Das combinações de parâmetros que testamos, `max_depth=9` com 200 árvores produz a melhor pontuação média de teste, e, se você procurar no DataFrame, verá que a ROC AUC = 0.776. Esse é o melhor modelo que encontramos até agora.

Em um cenário do mundo real, faríamos uma busca mais completa. Algumas próximas etapas interessantes seriam testar um número maior de árvores e não gastar mais tempo com `n_estimators < 200`, já que sabemos que precisamos de pelo menos 200 árvores para conseguir o melhor desempenho. Você poderia fazer a pesquisa em um espaço de `max_depth` mais granular em vez de pular três unidades como fizemos aqui e testar alguns outros hiperparâmetros como `max_features`. No entanto, para o livro, assumiremos que fizemos uma busca mais completa como essa e concluiremos que `max_depth=9` e `n_estimators=200` é o conjunto ótimo de hiperparâmetros.

Capítulo 6: Imputação de dados faltantes, análise financeira e distribuição para o cliente

Atividade 6: Derivando insights financeiros

1. Usando o conjunto de teste, calcule o custo de todas as inadimplências se não houvesse programa de aconselhamento.

Use este código para o cálculo:

```
cost_of_defaults = sum(y_test_all) * savings_per_default  
cost_of_defaults
```

A saída deve ser:

66308240.202088244

Figura A.19 – Custo de todas as inadimplências supondo ausência de aconselhamento.

2. Calcule em que percentual o custo de inadimplências seria diminuído pelo programa de aconselhamento.

A potencial diminuição no custo da inadimplência é a maior economia líquida possível proveniente do programa de aconselhamento, dividida pelo custo de todas as inadimplências na ausência de um programa:

```
net_savings[max_savings_ix]/cost_of_defaults
```

A saída é:

0.2329472975431598

Figura A.20 – Diminuição fracionária no custo de inadimplências que poderia resultar de um programa de aconselhamento.

Os resultados indicam que podemos diminuir o custo de inadimplências em 23% usando um programa de aconselhamento guiado pela modelagem preditiva.

3. Calcule a economia líquida por conta no limite ótimo.

Use este código para o cálculo:

```
net_savings[max_savings_ix]/len(y_test_all)
```

A saída deve ser:

2601.2673223171373

Figura A.21 – Economia líquida por conta possibilitada pelo programa de aconselhamento.

Resultados como esses ajudarão o cliente a dimensionar o possível

valor da economia que ele pode gerar com o programa de aconselhamento para qualquer que seja o número de contas fornecidas.

4. Plote a economia líquida por conta em relação ao custo do aconselhamento por conta para cada limite.

Crie a plotagem com este código:

```
plt.plot(cost_of_all_counselings/len(y_test_all), net_savings/len(y_test_all))  
plt.xlabel('Upfront investment: cost of counselings per account (NT$)')  
plt.ylabel('Net savings per account (NT$)')
```

A plotagem resultante deve ser esta:

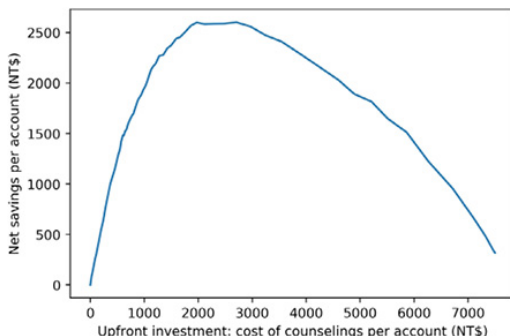


Figura A.22 – Custo inicial do programa de aconselhamento necessário para a obtenção de uma economia específica.

Isso indica quanto o cliente precisa orçar para o programa de aconselhamento em um mês específico para obter a economia desejada. Parece que o maior benefício pode ser criado por um orçamento de até NT\$2000 por conta. Depois disso, a economia líquida é relativamente pouca e então declina. Talvez o cliente não consiga reservar essa quantia para o programa. Porém, esse gráfico lhe dará provas para solicitar um orçamento maior se necessário.

Esse resultado corresponde ao nosso gráfico do exercício anterior. Embora tenhamos mostrado que o limite ótimo é 0.2, pode ser bom para o cliente usar um limite mais alto de até 0.25, fazendo então menos previsões positivas, oferecendo aconselhamento para menos titulares de contas e tendo um custo direto menor com o programa. A *Figura A.62* mostra como isso ocorre em termos de custo e economia líquida por conta.

5. Plote a fração de contas previstas como positivas (isso se chama “flag rate”) em cada limite.

Use este código para plotar a flag rate em relação ao limite:

```
plt.plot(thresholds, n_pos_pred/len(y_test_all))
```

```
plt.ylabel('Flag rate')  
plt.xlabel('Threshold')
```

A plotagem deve ficar assim:

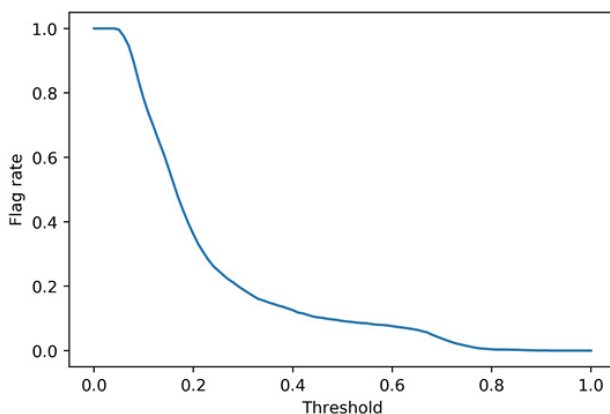


Figura A.23 – Flag rate em relação ao limite para o programa de aconselhamento de crédito.

Essa plotagem mostra a fração de pessoas previstas como inadimplentes e, portanto, para as quais é recomendada assistência em cada limite. Parece que, em um limite ótimo de 0.2, só cerca de 30% das contas serão marcadas para aconselhamento. Isso mostra como usar um modelo para priorizar contas para aconselhamento pode ajudar a encontrar as contas certas e reduzir o desperdício de recursos. Limites mais altos, que podem resultar em uma economia quase ótima até um limite de cerca de 0.25 como mostrado na *Figura A.36 (Exercício 25, Caracterizando custos e a economia)*, resultam em flag rates mais baixas.

6. Plote uma curva precision-recall para os dados de teste usando o código a seguir:

```
plt.plot(n_true_pos/sum(y_test_all), np.divide(n_true_pos, n_pos_pred))  
plt.xlabel('Recall')  
plt.ylabel('Precision')
```

A plotagem é mostrada na Figura A.24.

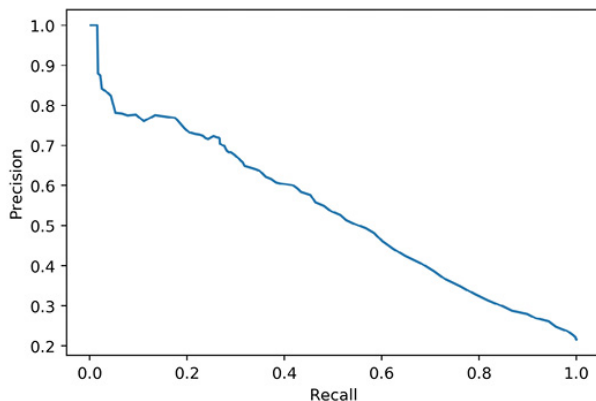


Figura A.24 – Curva precision-recall.

A *Figura A.24* mostra que, para começar a obter uma taxa de verdadeiros positivos (isto é, recall) muito acima de 0, temos de aceitar uma precisão perto de 0.75 ou mais baixa. Logo, parece que há espaço para melhoria em nosso modelo. Embora isso não precise ser informado ao cliente, mostra que com o uso de técnicas de modelagem mais avançadas, ou um conjunto de características mais rico, o desempenho do modelo pode ser melhorado.

A precisão e o recall têm uma ligação direta com o custo e a economia do programa: quanto mais precisas forem suas previsões, menos dinheiro você perderá em aconselhamento devido a previsões incorretas. E, quanto mais alto o recall, mais economia poderemos gerar identificando com sucesso contas que ficariam inadimplentes. Compare o código dessa etapa com o usado para calcular o custo e a economia no exercício anterior para ver. Isso liga a análise financeira às métricas de machine learning que examinamos no início do estudo de caso.

Para ver a conexão da precisão e do recall com o limite usado para definir previsões positivas e negativas, pode ser instrutivo plotá-los separadamente.

7. Plote a precisão e o recall separadamente no eixo y em relação ao limite no eixo x.

Use este código para produzir a plotagem:

```
plt.plot(thresholds, np.divide(n_true_pos, n_pos_pred), label = 'Precision')
plt.plot(thresholds, n_true_pos/sum(y_test_all), label = 'Recall')
plt.xlabel('Threshold')

plt.legend()
```

A plotagem deve ficar assim:

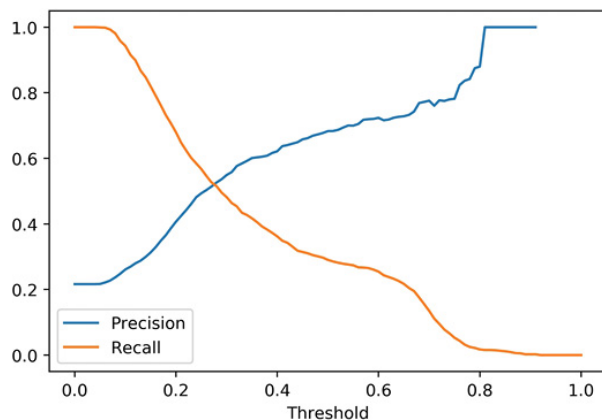


Figura A.25 – Precisão e recall plotados separadamente em relação ao limite.

Essa plotagem explica por que o limite ótimo é 0.2. Embora o limite ótimo também dependa da análise financeira de custos e economia, podemos ver aqui que a parte mais íngreme do aumento inicial na precisão, que representa a exatidão das previsões positivas e, portanto, é uma medida de como o aconselhamento guiado por modelo pode ser eficaz, ocorre até um limite de cerca de 0.2.